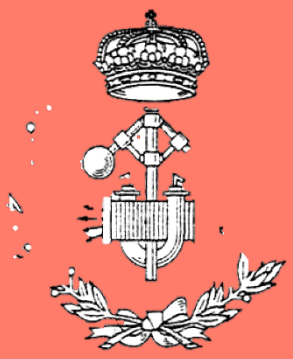
E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Control of a Single Degree of Freedom Helicopter



Grado en Ingeniería
en Tecnologías Industriales

Trabajo Fin de Grado

Irene Miquelez Madariaga

Director: Jorge Elso Torralba

Pamplona, 1 de julio de 2016.

upna Universidad Pública de Navarra

Contents

List of Figures

List of Tables

Abstract

The current thesis continues with the projected about the control of a mockup with a single degree of freedom started by Azcona [1]. This same objective is addressed from a new perspective, as the transfer function will be obtained by the identification of the plant, based on input-output data obtained from different tests, instead of applying basic physical laws. In order to reach the objectives, the first step is to obtain the open loop functioning of the system. To this end, it is necessary to include a new encoder, that meets the needs for sufficient resolution and suitable requirement of reading speed of the problem. Besides, the different microcontrollers offered in Arduino boards were considered, in order to see which of them is more appropriate for the requirements of the posed problem. Additionally, a coder that allows a correct reading of the encoder, the generation of a PWM signal with a high enough resolution and the exact sampling of input-output signals is developed. After performing the identification, different controllers were proved in a closed loop configuration. It is possible to observe the variations in oscillation and system velocity depending on the position, validating the expected non linearity of the system.

Key words: control, Arduino, PWM, encoder.

Resumen

El presente proyecto continua el trabajo iniciado por E. Azcona [1] sobre el control de una maqueta de un grado de libertad. Dicho objetivo se aborda desde una nueva perspectiva, ya que la función de transferencia se obtiene mediante la identificación de la planta, con datos de las señales de entrada y salida, resultado de diferentes ensayos, en lugar de la aplicación de leyes físicas. Para lograr dichos objetivos se trabaja en primer lugar el funcionamiento en lazo abierto del sistema. Para ello se realiza el cambio del sensor y se emplea otro cuya resolución y requerimiento de velocidad de lectura se adecuen a las del problema. Además, se estudia qué microcontrolador, entre los ofertados en las placas Arduino, ofrece unas prestaciones que se adecuen a las necesidades del problema planteado. Por otra parte, se desarrolla un código que permita la correcta lectura del encoder, la generación de una señal PWM con una resolución suficiente y el muestreo exacto de las señales de entrada y salida. Tras la identificación de la planta, se comprueba el comportamiento del sistema en lazo cerrado y la respuesta obtenida con el uso de diferentes controladores. Se observa la variación en las oscilaciones y la velocidad del sistema en función de la posición, quedando verificada la no linealidad del mismo.

Palabras clave: control, Arduino, PWM, encoder.

Chapter 1

Introduction

1.1 Objectives

The overall objective of the project is demonstrating the feasibility of a control for the model with simple tools. It will be achieve by fulfilling the following steps:

- Solving the problems concerning the correct operation of the different parts of the system, particularly the measurement of the encoder and the generation of PWM wave.

- Performing the system identification for the validation of the mathematical model. This will be achieved through the design of the identification test for obtaining data and its posterior analysis.

- Desing and implementation of a digital controller in an Arduino board.

1.2 Scope

This project aims to continue the task started in a previous one by controlling a one degree of freedom structure with an Arduino Uno microcontroller. The previous work contained a theoretical calculation of the systems transfer function by linearization of the expressions obtained from physical laws and a PI controller based on the transfer function.

Although all elements necessary for a closed loop operation were calculated and assembled, the objective was not achieved, therefore the fist objective of the present project is to obtain an open loop operation in order to see which are the requirements of the sensor and the motor.

The first part of the present project studies the characteristics of the different components of the system and shows the connexion between them. The following chapter (3) shows the basic structures and tools of the microcontroller used in the code in order to read the sensor signal and generating a PWM signal such as Timers and Interrupts.

Chapter 4 contains open loop code and different identification tests with the transfer functions obtained form the measured data.

Finally, chapter 5 presents different PI controllers, closed loop code and a comparison between measured results and theoretical simulations.

Chapter 2

Description of the components

2.1 Metallic structure

The plant consists on a metallic structure made of an aluminum-steel alloy (Azcona, 2014, [1]). The two main parts are a vertical piece, with an L-shape that easies the fastening to a table, and a thinner bar that plays the role of the moving part of a rocker.

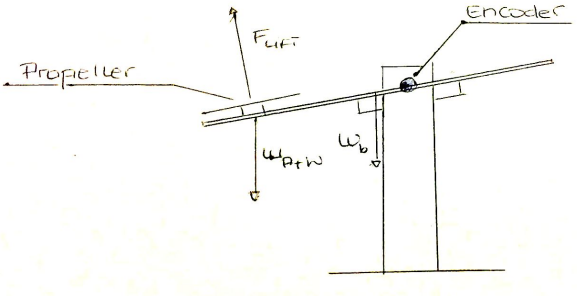


Figure 2.1: Plant draft

Both parts are joint through a shaft that turns in conjunction with the encoder

shaft. In order to ensure that no slide occurs, the shafts have been stuck together with an adherent filler. The shaft does not cross the middle point of the horizontal bar and beside the motor (45g.) there is 15g. extra weight. This implies that the external force supplied by the propeller is necessary to balance the bar.

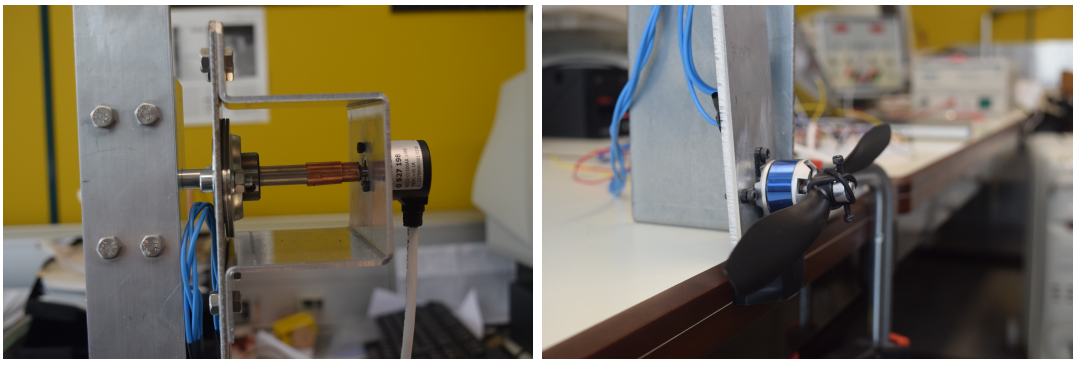


Figure 2.2: (a)Shaft joint, (b) Propeller

2.2 Motor

Brushless motors are becoming a widely used tool for propulsion in model aircraft applications. Opposing to traditional gas propellers, brushless motors are smaller, simpler, cheaper, cleaner and less noisy, perfectly fitting into laboratory applications. Besides, having a fixed model solves the main drawback of electrical propulsion, which is short flight times due to the need of batteries. In this case, a constant voltage source is used.

Figure 2.3: BL2210 Brushless motor [2]

Brushless motors are synchronous motors with permanent magnets which do not require brushes for introducing the current in order to create a magnetic field. The most general characteristic that needs to be taken into account is the propulsion, that is related to the power of the motor and the diameter of the helix and depends on the weight of the model. When choosing a motor, another important parameter is the rpm per volt. The model requires a BL2210/30 motor (Figure 2.3) with a 10cm radio propeller.

Azcona (2014, [1]) uses Servo library for Arduino in order to control de motor. Servo are DC motors whose principal characteristic is the possibility of being immediately positioned to any angle within their range.

Servo library uses a 500Hz PWM signal for selecting the desired position for the servo. A 50 % duty cycle indicates a 0 degrees position and a 100 % corresponds to 180 degrees (or the corresponding full range). In order to make the motor rotate, the value of the duty cycle should be changed continuously, although the movement will only occur within the allowed scope.

However, the motor used for the project is a brushless motor with an Electronic Speed Control device. The ESC system requires a constant 12V power signal and a 500Hz 0/5V PWM control signal. The motor with the helix will rotate at a speed

proportional to the amplitude of the power signal, meaning that it will also be proportional to the duty cycle of the PWM signal.

The motor used for this application allows turning in a single sense which could be invertid by exchanging two of the phases. This is the same as saying that it is not possible to have a negative PWM input to the converter.

The role of the motor in the system is vital, as it will allow to reach stable points at angles different from 0^o. The lift force made by the propeller could be simplified to the following expression:

$$F_{Prop} = \frac{1}{2} A \rho C_L R^2 \omega^2 \tag{2.1}$$

where ρ corresponds to air density, C_L is lift coefficient, R is the radius of the propeller and A is its area. ω represents the angular speed of the propeller and will be the only changing parameter. It will vary approximately proportionally to the duty cycle. However, it is convenient to take into account, that the influence of the lift force will also depend on the position of the bar, as for small angles, the greatest component of the force will be in the *horizontal direction* and for angles close to 90^o the force will be mostly affecting to the *vertical direction*.

2.3 ESC

Electronic Speed Control is the electronic circuit in charge of varying the motor velocity. The inputs of the ESC are a DC voltage (power signal, 13,8V) and a PWM or control signal that sets the switching rate of the transistors in the ESC and the velocity of the motor. The output is a three-phase alternating current that can

resemble a sine or be trapezoidal.

Figure 2.4: Simon K Series 30A ESC [3]

A typical value for the PWM period in model aircraft applications is 2ms (500 Hz). Depending on the model, having the possibility of rotating the propeller in both senses can be useful. For that purpose, it is necessary to change the polarity of the phases and all duty cycle values between 0 and 100 are used. However, in this project just one rotation direction is needed and the motor stars moving with a 50% duty cycle approximately.

When choosing the ESC, its power, voltage and current requirements should satisfy and improve the ones from the motor. The selected ESC is Simon K Series 30A (Figure 2.4).

2.4 Encoder

Encoders are sensors used to measure the number of rotations, the angle or the linear movement of physical systems. Regarding the constructive technology, rotatory encoders can be:

Conductive: They consist of metallic contacts at different distances to the axis. Depending on the position the can be touching a metallic disc allowing electric

current to flow, or will be standing in a gap in the disc, preventing the current to go through them.

Optical: Rotating discs incorporate transparent and opaque slots and a light source. A light detector will read either a logic zero or one, depending on the position of the encoder.

Magnetic: They use magnetoresistance or Hall effect to track position with a series of magnetic poles.

Capacitive: with the use of asymmetrical discs, the capacitance changes depending on the angle.

Based on the measure obtained with the sensors encoders are classified as incremental or absolute.

Incremental: Their application depends on the number of channels they have. Single channel encoders can only measure the presence of motion whereas dual channel incremental encoders can detect motion and rotation direction. There could also be a third channel with a slot per turn in order to have a reference. Incremental encoders are most economical and used for many purposes.

Absolute: They have a distinctive digital output for each position and can be divided depending on the type of codification they use. The most typical ones use Gray code, for as it easies error correction.

The selection of the encoder is an important point of the project for they are usually an expensive component and more subject to introducing errors in the system than other parts. For this case, an incremental encoder is good enough as the starting position will always be the same.

2.4.1 Quadrature Encoder

Quadrature encoders are digital transductors which use two circles of slots in order to keep track of the movement and the direction of the rotation. Being incremental encoders means that the angle position reference must either be set by software or the program must always be initialized in the same position.

Figure 2.5: Icuro IR32 [4]

Although the resolution of the encoder is usually given by the number of slots per turn, taking into account all the possible combinations (channel A: 0 or 1, channel B: 0 or 1), resolutions of 2 or 4 times the number of divisions per turn can be achieved.

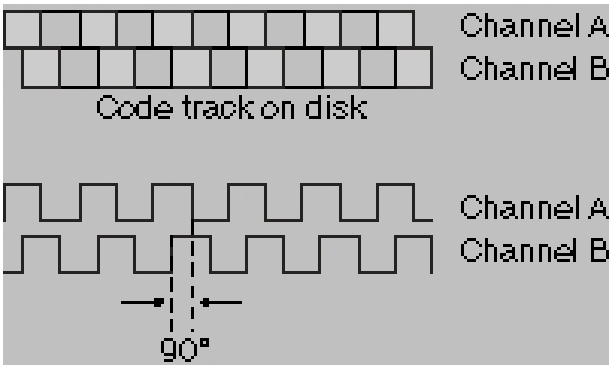


Figure 2.6: Quadrature encoder slots distribution [6]

x1 resolution

If there is a call to the Interruption function every time the channel A pin detects a rising edge, it will be invoked n times/turn (being n the number of slots/turn). In this case channel A will always be a logic one and therefore, only channel B input pin must be checked.

As figure 2.6 shows, if after detecting a rising edge in channel A, channel B is low, the movement direction is towards right. On the other hand, if it is high, the shaft is turning left. Assuming positive movement to the right, ChannelB a digital input and Pulse a global integer variable, the resulting code would be the following.

```
1
2 Encoder void(){
3     if (ChannelB==1){
4         Pulse--;
5     }
6     else{
7         Pulse++;
8     }
9 }
```

x2 resolution

If instead of detecting only rising edges of channel A, there were a call to the interrupt every time there is a change in the pin level, there would be 2*n divisions for the full turn and the measure would be twice as accurate. Besides changing *attachInterrupt* to *CHANGE* mode, interrupt code must also be adapted.

By looking at figure 2.6 it is possible to notice that after every level change in

channel A, the following table holds:

Channel A	Channel B	Pulse
0	0	+1
0	1	-1
1	0	-1
1	1	+1

Table 2.1: x2 resolution

Consequently, the resulting code is:

```
Encoder void(){
    if (ChannelB==ChannelA){
        Pulse++;
    }
    else{
        Pulse--;
    }
}
```

Figure 2.7: Encoder: x2 resolution

x4 resolution

In order to obtain four times the number of slots resolution, every edge in both channels must be detected. Resultantly, the two external interrupts existing in Arduino Uno must be used. The code used for reading the encoder must satisfy this table:

Detected edge in channel	Channel A	Channel B	Pulse
A	0	0	+1
A	0	1	-1
A	1	0	-1
A	1	1	+1
B	0	0	-1
B	0	1	+1
B	1	0	+1
B	1	1	-1

Table 2.2: x4 resolution

The code used for implementing this logic function is:

```
EncoderA_void(){
    if (ChannelB==ChannelA){
        Pulse++;
    }
    else{
        Pulse--;
    }
}

EncoderB_void(){
    if (ChannelB==ChannelA){
        Pulse--;
    }
    else{
        Pulse++;
    }
}
```

Figure 2.8: Encoder: x4 resolution

The selection of the code will depend on the initial resolution of the sensor, the measurement requirements of the project, the processing speed of the microcontroller

and the complexity of the other parts of the code.

2.5 Arduino

Microcontrollers are integrated circuits which comprise every functional block of a microprocessor in a single encapsulation. They receive, interpret and generate external and internal digital signals and appear in control applications. This project wants to prove the feasibility of control strategies implemented in Arduino Boards (Atmel). The most important advantages provided by Arduino products are:

- Open-source easy to use software.
- Compatible and specially prepared tools to use with Arduino Boards including industrial and academically used softwares such as Matlab and Labview.
- Affordable prices and accesible components
- Well known product widely used in amateur applications. As a consequence there is plenty of accessible information in Internet.

There exist various Arduino Board models with different microcontrollers and providing diverse features. In order to choose which one is the most appropriate for the given application, it is necessary to evaluate which one covers the requirements and permits the easiest code. Only Arduino Uno and Arduino Due are going to be considered as they are the most accessible ones in this case.

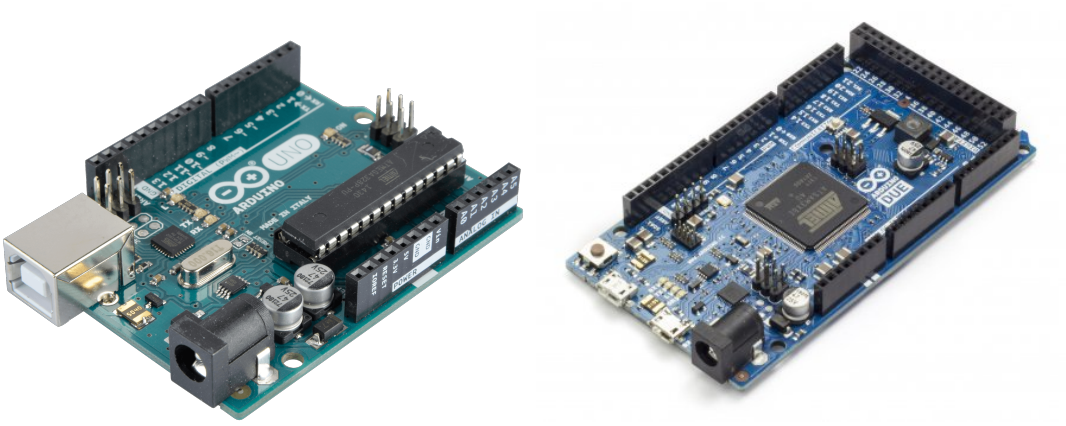


Figure 2.9: (a)Arduino Uno and (b) Arduino Due [8]

2.5.1 Arduino Uno

Based on Atmega328P, it holds 14 digital I/O pins, 6 PWM outputs, 6 analog inputs and a 16MHz quartz crystal. It is the simplest Arduino Board and it is possible to find many examples of applications using it in internet. Its main **advantages** are:

- 500 Hz PWM output, which means there is no need to change the frequency, as it matches the one required by the ESC device.
- It is one of the cheapest boards.
- 5V technology, just as the sensors output and the ESC input. No need to converse the voltages.
- Atmega328P microcontroller is easy to program even using direct C code.

Arduino Uno presents the following **drawbacks**:

- The internal clock frequency is 16MHz, which could be too slow for certain instructions.

- Arduino Uno is a simple board with fewer implemented functions, libraries and features. This means that for obtaining a more flexible use of PWM and other timer options, it is necessary to use ***c-AVR*** to program the microcontroller.

2.5.2 Arduino Due

Arduino Due supports an Atmel SAM3x8E ARM Cortex-M3 CPU. The board supplies 54 digital I/O pins, 12 PWM outputs, 12 analog inputs and a 84MHz clock. The most important benefit of using an Arduino Due board is the possibility to use the instruction *analogWriteResolution()* which allows to set the PWM duty cycle resolution between 8 and 12 bits.

Nevertheless, using Arduino Due implies facing two significative disadvantages:

- Arduino Due works with 3.3V technology. As a consequence, a way to convert the 5V signal coming from the encoder to 3.3V and then the PWM output back to 5V would be needed.

- PWM output frequency is 1000Hz, which does not work so well in the electronic speed controller. This problem could be avoided by programming the microcontroller timer instead of using *AnalogWrite* function.

2.6 Assembly

Figure 2.10 depicts the assembly of all system components:

- Microcontroller

- Motor+propeller

- ESC
- Encoder
- Potentiometer
- Voltage source
- Computer

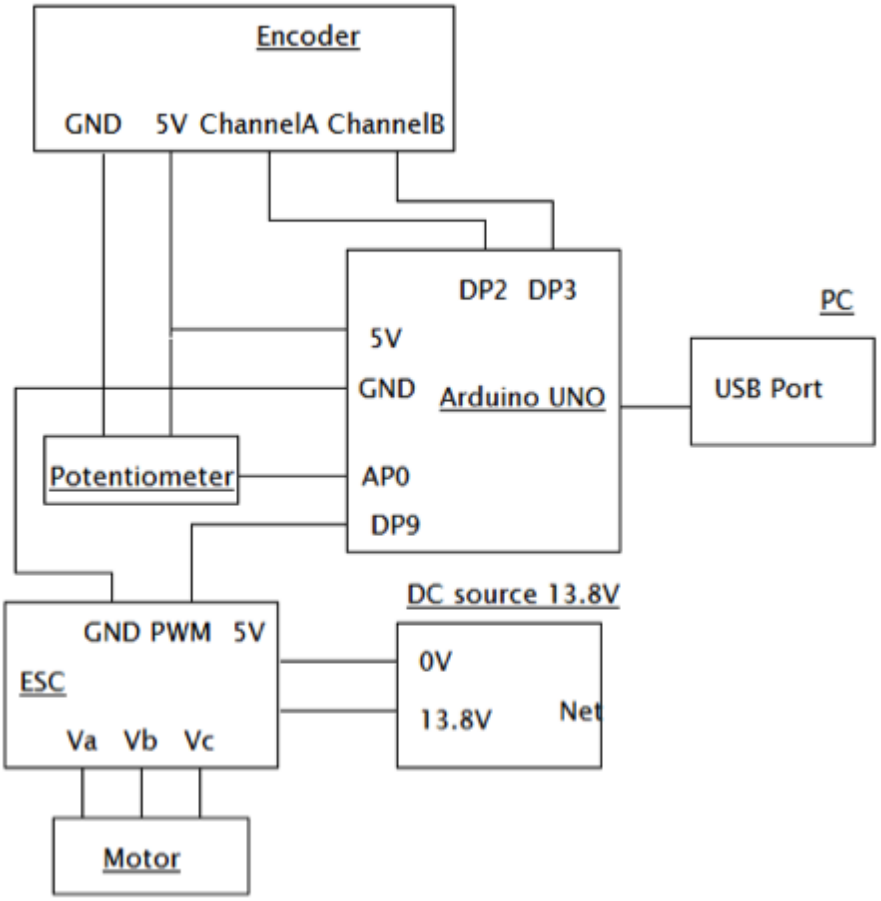


Figure 2.10: Assembly schematic

Digital pins 2 and 3 must necessarily be devoted to the encoder because they are

the only ones supporting external interrupts. Besides, digital pin 9 is the output for the PWM wave generated by timer 1.

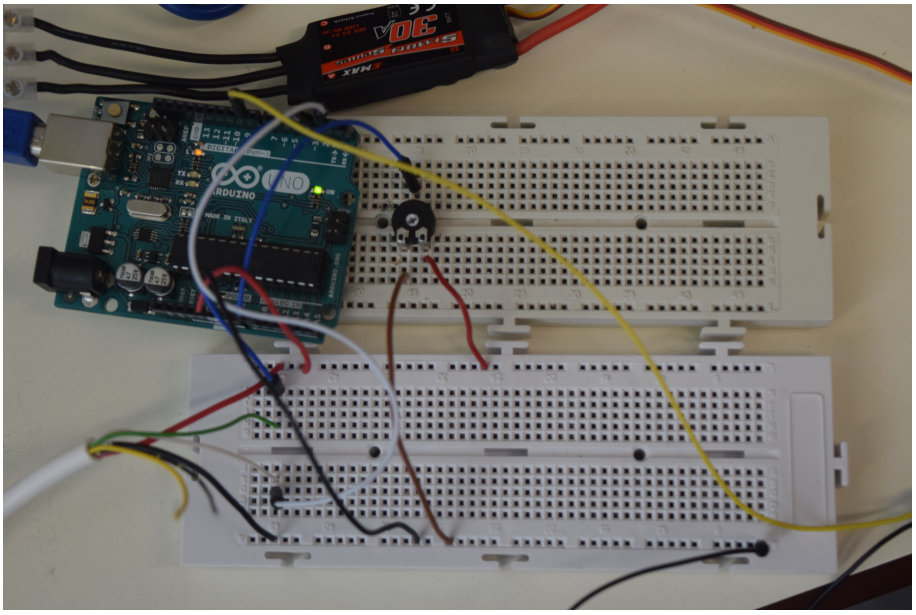


Figure 2.11: Device assembly

Chapter 3

System operation

3.1 Interrupts

Interrupts bide external events and permit the microprocessor to establish connection with other physical elements (peripherics). Interrupt functions simplify the incorporation of real time systems and the distribution of the code. They work with the same procedure than subroutines or functions, but instead of being called to with a line of code or a command, they are activated by a hardware mechanism.

The different possible ways to invoke an interrupt function are hold in table 3.1.

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Figure 3.1: Reset and interrupt vectors [7]

This means interrupt functions can be called by a reset, a level change in certain pins, timers 0, 1 and 2 in their different modes, the watchdog timer, serial port and other modules. This table also shows the priority in the performance of the interrupts if two of them are invoked simultaneously (the lower the vector number, the higher the priority).

As interrupts are usually needed in order to perform tasks that must be instanta-

neous, it is convenient to define the involved variables as *volatile*. Volatile variables are stored in RAM instead of in a storage register for a more reliable use, when their value does not only depend on the processor but on external events.

ISR functions should not return any parameters and the code inside them must be kept as short and simple as possible. Besides, other functions such as *millis()* or *delay()* could not work as expected inside and interrupt function.

For the present application, timer overflow (with timer 2 in compare mode) and Pin change interrupts are needed.

3.1.1 External Interrupt

Arduino Uno board counts with two digital pins, 2 and 3, enabled to invoke external interrupts (*INT0* and *INT1*). The functions can be called by rising or falling edges and a low level input.

In Arduino programming environment, *AttachInterrupt(digitalPin, ISR, mode)* function is used in order to include an interrupt function. The parameters inside the functions are:

digitalPin: In Arduino Uno digital pins 2 and 3 are available.

ISR: This parameter sets the name of the function, which must be defined afterwards.

Mode: which could be *rising, falling, change, or low.*

Although for reading the new encoder **.ino** functions are fast enough, knowing how the microcontroller should be programmed could be interesting in order to fully understand how interrupts work.

Interrupt mode is specified by the External Interrupt Control Register (**EICRA**). Bits 3 and 2 select the mode of INT1 and bits 1 and 0 select the mode of INT0 [Table 3.1].

ISCx1	**ISCx0**	**Mode**
0	0	LOW
0	1	CHANGE
1	0	FALLING EDGE
1	1	RISING EDGE

Table 3.1: External interrupt mode.

When bits 1 and 0 of in External Interrupt Mask Register (**EIMSK**) are set to 1, they enable interrupts 1 and 0 respectively. The remaining 6 bits of the register are always read as 0.

Bits 1 and 0 of External Interrupt Flag Register (**EIFR**) contain the interruption flags. When the interrupts are requested, the corresponding flag is set to 1 and then cleared after the function is executed.

3.1.2 Timer Overflow Interrupt

As said before, timers are other activation mechanism for interrupt functions. The mechanism for calling the function can be different depending on the timer mode. Timer 2 in *Clear Time on Compare Match* is used for an accurate sampling time of 20 ms. The 8 bit timer 2 counter increments its value every operation cycle until it equals the value set in **OCR2A**.

Once that value is reached, the interrupt function is read, the timer gets reset to cero and is incremented to the value of **OCR2A** again.

3.2 Timers

Microcontrollers require the use of a clock for being able to perform synchronized operations. This clocks could be external, connected to a dedicated input, or internal, as happens in *Atmega328p*. In this case, the clock is internal and oscillates at 16MHz frequency.

Timers are associated modules that count in synchronism with the clock. The maximum value they can count up to depends on the number of available bits and the possible values for the pre and postscalers. Arduino Uno includes three timer modules (0, 1 and 2). Timer 0 and 2 are associated to 8 bit registers and, as a consequence, reach a maximum value of 255. Timer 1, being a 16 bit counter, has a maximum value of 65535. On addition to this, prescaler takes one of the following values: 1, 8, 64, 256 or 1024. This means that the longer period which timer 1 counts up to is 8388.608 ms.

Timer configuration allows to select among diverse operation modes. The difference between them lies in the signal generation and compare output mode configuration. There are a total of 15 different modes for *Atmega328p* but only the interesting ones for the application are explained in detail:

Fast PWM mode: with **ICR1** as TOP value. This mode provides a simple way to program a PWM output. As the application requires high resolution, timer 1 is used.

Register **ICR1** sets the signal frequency through the following expression:

$$T_{PWM} = \frac{1}{16 * 10^6 Mhz} * prescaler * (ICR1 + 1) \tag{3.1}$$

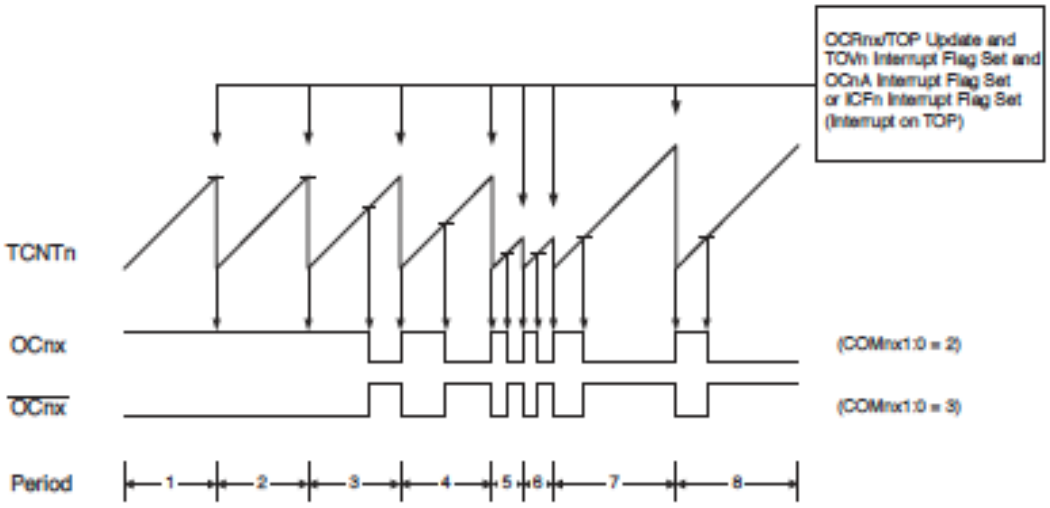


Figure 3.2: Fast PWM timing diagram [7]

with $T_{PWM} = 0.002s$, $prescaler = 1$ and $ICR1 = 31999$.

Duty cycle is given by register **OCR1A** (0-65535). When **TCNT** matches **OCR1A** the output value changes from *HIGH* to *LOW* and the counter continues increasing its value until it reaches the one stored in **ICR1**. Then it is reset to 0 and a new period begins.

Clear time on compare match: **TCNT2** register increments its value until it matches the one stored in **OCR2A**. It is possible to enable an interrupt function which would be invoked every time this happens. If **OCR2A** is actualized to one lower than **TCNT2** the interrupt will be missed.

When modifying the registers that define timer modules, one must take into account that certain functions in Arduino libraries could be modified. Using functions such as *delay()* or *millis()* is inadvisable.

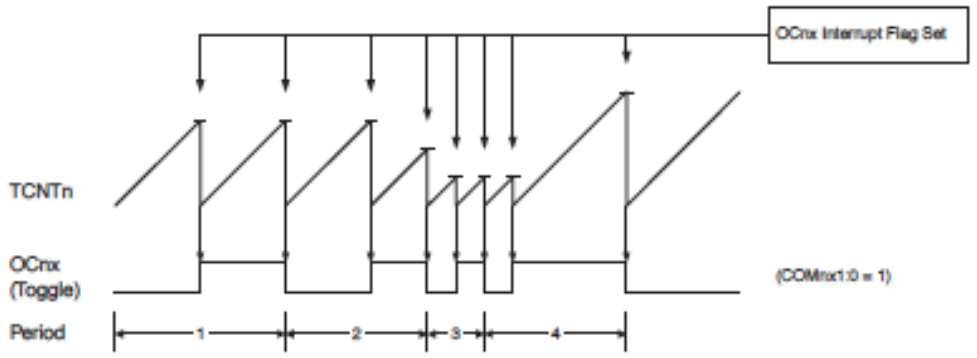


Figure 3.3: Clear time on compare match timing diagram [7]

3.3 Sensor reading

Taking Azcona [1] as a starting point, the studied mock up includes a Hengstler quadrature encoder (2.5) with 1000 per turn. Whilst implementing the first controller, it was observed that the control action did not have the expected value. After discarding saturation problems (demanding duty cycles out of the 0 to 255 range allowed by *analogWrite* function) by introducing limits in the code and numeric errors at computing the values, the only possible problem was a mistaken read of the position.

The first approach was to isolate the code devoted to reading the encoder. This first code obtains a x1 resolution of the encoder and has the simplest possible structure:

```
1 int PinA=2;
2 int PinB=4;
3 volatile int Pulse=0;
4 int sentido;
5 int Angulo, val;
```

```
6
7  void setup() {
8    Serial.begin(9600);
9    attachInterrupt(0, Encoder, RISING);
10   pinMode(PinA,INPUT);
11   pinMode(PinB,INPUT);
12 }
13
14 void loop() {
15   Serial.println(Pulse);
16 }
17
18 void Encoder() {
19   sentido=digitalRead(4);
20   if (sentido==0){
21     Pulse++;
22   }
23   else Pulse--;
24 }
```

Some likely explanations for the origen of the error were the following:

- Delay in communication with the serial port. *Serial.println(Pulse)* function takes more clock cycles that other functions such as *digitalWrite()*. Besides, it is inside *void loop*, therefore it has a lower priority than the interrupt. As a consequence, it could happen that the edges arrive from the encoder with a higher speed than what it takes to establish serial communication and the information was not updated on time. However, even if it is not possible to see it, *Pulse* counter should have been updated correctly.

To ensure that this was not the problem, the system was manually move from the rest position and then taken back to it. As it is an incremental encoder, a correct reading would imply having counter *Pulse* back to cero when taken to the initial position. Nevertheless, after giving serial communication enough time to update the latest value, the angle did not match the expected value. It could be deduced that there were pulses being lost.

- One of the most common problems when reading quadrature encoders and other sensors that have a square signal as an output is the degradation of the pulses. If instead of having well defined steps, the output resembles an exponential decay, the microcontroller might not identify the falling edges and, as a consequence, miss samples. This option was discarded by observing the signal in an oscilloscope and verifying that it rapidly reaches the limit the microcontroller sets for *HIGH* and *LOW* digital input values.

- Other, though less likely, possible source of error was a conflict in the priority of the code. Nevertheless, this should have been solved by using external interrupts, as they have the highest priority.

- The last considered option was that interrupt function execution time lasts longer than the interval between two rising edges in the signal coming from the encoder. This would imply an inevitable sample lost, as once the function has been invoked, the interrupt flag will be enabled until the whole code inside it has been read.

By means of function *millis()*, which gives the time in milliseconds that has gone by since the program was compiled, it is possible to measure the time between to

consecutive interrupts. With the simplified code, this interval is around 160μs.

Bearing in mind that a 90 degrees angle corresponds to 250 slots, this would mean that the maximum admissible speed is 0.25 turns/s, which is far to slow for the system.

The first considered solution was the optimization of the code by programming directly in ***AVR-c***. Arduino code is based on libraries that are compatible with Atmel microcontrollers. This implies that the code is not always optimized (the microcontroller reads orders that are not strictly necessary) and there is not a direct control over what happens in the registers as does when programming with ***Assembler*** . For instance, as explained in [5], *void loop* does not really work as a *while()* loop but takes four times what it takes *while()* to perform a simple operation. Among other functions, *void loop* checks the serial configuration adding orders to the ones written by the user.

Until now, the program was based on Arduino functions, but implementing the code using ***AVR-c*** could make a difference in certain in certain parts of it. For example, reading a digital input could be 50 times slower using *digitalRead()* than directly reading the I/O port with *val = 0x01&(PIND)* (reads bit 0 in port D).

This option could be interesting, though it makes programming far more complicated than just using ***Arduino*** code.

Other possible option could be using an special IC devoted to counting input signals such as the one coming from the encoder, but it would mean programming it and converting its output to data that fits into the code.

The adopted solution was using an encoder with lower resolution. The chosen model was similar to the previously used, but with 100 slots/turn instead of 1000.

This way, as explained in chapter 3 it is possible to get 100, 200 and 400 pulses/turn resolutions. By performing the same tests as to the first encoder, it was proved that no pulse was missed.

3.4 PWM

Pulse width modulation or PWM is a way of obtaining analog control signal through a square signal or constant period. **Ton** is defined as the timer the signal has a high logic level and **Toff** as the time the signal has a low logic level. So that period (**T**) holds:

$$T = Ton + Toff \tag{3.2}$$

Duty cycle is defined as:

$$DC = \frac{Ton}{T} \tag{3.3}$$

By varying the duty cycle of the PWM signal sent from the microcontroller to the ESC device, the amplitud of the wave that fuels the motor changes and so does its rotating speed.

Manual PWM

There are many possible options for generating a PWM wave with Arduino but the easiest one would be manually changing the level of a digital output.

```
1  void setup() {
2    pinMode(13, OUTPUT);
3  }
4  void loop() {
5    ton= map(analogRead(0), 0, 1023, 0, 2000);
6    toff=(2000-ton);
7    digitalWrite(13,HIGH);
8    delay(ton);
9    digitalWrite(13,LOW);
10   delay(toff);
11 }
```

Figure 3.4: Manual PWM.

In this code, Ton is directly selected with a potentiometer by reading an analog input. Although this method could work if the code is kept simple, it should be taken into account that the pin will not stay in the desired level just the time chosen in the *delay()* function, but until a new *digitalWrite()* function appears. This means that any instruction read between them will distort the time intervals and would make the real duty cycle deviate from the desired one. If the code has logic statements that lead to different lines (if or while sentences), the deviation would not only exist but also differ from one period to the other, therefore changing the frequency slightly. In this past example, this would not be critical, but introducing new parts of the code would make it useless.

Timer interrupt

Using timer interrupt functions makes it easier to ensure that the PWM period is held constant. Changes in the digital output would be performed inside the ISR (interrupt service routine). As Ton will start every time the function is invoked, the period of the wave will be constant if there is no other interrupt function of higher priority. In this case, the external interrupt function that reads the encoder output has a higher priority, so it would not be possible to have a precise control on the signal.

```
#include <avr/interrupt.h>
int  cont, val;
void setup() {
    Serial.begin(9600);
  pinMode(5, OUPUT);
  //REGISTERS AND BITS ASSOCIATED TO TIMER 2
    cli();// Deactivating global interruptions
      TCCR2A = 0;
      TCCR2B = 0;
      TCCR2A =  (1<<WGM21);// Compare mode
      OCR2A = 31;
      TIMSK2 = (1 << OCIE2A);// Activating interruption
      TCCR2B = 0x07;// Prescaling at 1024
       sei(); //Global interruption activation
}
void loop() {
    val=analogRead(0);
    Ton=map(val,0, 1023, 0, 255);
```

```
19 }
20 ISR(TIMER2_COMPA_vect) {
21       digitalWrite(5, HIGH);
22    delay(Ton);
23    digitalWrite(5,LOW);
24  }
```

analogWrite() function

Arduino programming environment offers the function *analogWrite*, which generates a 500Hz PWM wave and has an 8 bit resolution for the duty cycle. The main advantage of this instruction is that the wave will have the aim duty cycle and exact frequency until a new *analogWrite* order changes the DC. In other Arduino boards such as Arduino Due, it is possible to adjust the resolution through the command *analogWriteResolution*, increasing it to 12 bits.

Timer 1 in fast PWM mode

Another approach, which allows to achieve a more direct control on the PWM signal, is to program one of the timer modules of the Atmega328P microcontroller in its *fast PWM* mode. As we seek for a better resolution, it is necessary to use **Timer1**, as it is the only module in Arduino Uno with a 16 bit counter. The overall functioning for the fast PWM mode is described in section 3.2.

There are several registers that should be defined in order to use the counter as a PWM module and also to select frequency and duty cycle. Their name and configuration may change depending on the selected counter module. The ones described hereafter correspond to timer 1.

Registers **TCCR1A** and **TCCR1B** set the configuration of the module.

Register **TCCR1A**:

Bit 7:6=10, COM1A: Channel a in compare output mode, fast PWM.

Bit 5:4=00, COM1B: Channel B deactivated.

Bit 3:2=00: not implemented.

Bit 1:0=10, WGM 11,10:

Register **TCCR1B**:

Bit 7=0, ICNC1: noise filter deactivated.

Bit 6=0, ICES1: unimplemented for fast PWM mode.

Bit 5=0: reserved bit.

Bit 4:3=11, WGM 13, 12: selecting fast PWM mode.

Bit 2:0=001, CS 12,11,10: Internal clock without prescaler.

ICR1 is used for setting the frequency according to the following expression:

$$f_{PWM} = \frac{f_{osc}}{N * (1 + ICR1)} \tag{3.4}$$

where N is the value for the prescaler and equals 1.*f_osc* is 16MHz in Arduino Uno and *fPWM* is set to 500Hz. The result of the operation is that ICR1 should equal 31999.

Register **TCNT1** keeps track of the count, so is set at 0 in the setup loop. Duty cycle is written in the 16 bit *OCR1A* register, so that it can take vales between 0

and 65535. Writing a value in said register works exactly as an *analogWrite* order, meaning that the output signal will not change until a new order of changing the DC or deactivating timer 1 is read.

3.5 PWM resolution

Arduino One offers an 8 bit resolution for the PWM output using the function *analogWrite*, this means there are 256 possible values for the duty cycle. Nonetheless, there is a minimum value of the duty cycle for the motor to start rotating and a higher limit in the stable zone, so the input range is significantly reduced.

This means that only few angle values can be reached exactly. If the reference is set somewhere between two of this values, the system will oscillate between the upper and lower duty cycle values, reaching a limit cycle.

In order to solve this problem several options were considered.

As said before, Arduino Due allows PWM resolution to be easily changed from 8 to 12 bits. On the other hand, using the new board involves facing new problems.

The voltage incompatibility between the sensor, the ESC and Arduino was easily solve, though the signals speed had to taken into account.

The **encoder output** is a 5V/0V square wave. As the interest is set on counting the rising and falling edges, there should not be greater error that the admitted by the voltage range Arduino has for low and high value inputs. The simplest solution means using a voltage divisor which will not distort the signal speed. Instead of settling for fixed resistors, a 1kΩ was used for each of the encoder channels. This way the impedance needed was set manually.

PWM output is a more critical point, as ESC device is more sensitive to the

control input and it is very important that it can reach the full 0V to 5V range in order not to lose duty cycle resolution.

The first attempt consisted on using an AND gate with the following logic table:

Input A	Input B	Output
0	0	0
0	1	1
1	0	0
1	1	1

Table 3.2: Gate and

The control signal will enter both inputs, so only the first and last case should be considered. As is clear from the table, the output matches the input but with 5V instead of 3.3V by reason of the logic technology used for the gates. This is a very simple assembly which should grant the tracking of the PWM signal. However, due to the losses in the transistors, it would only reach 4.2V which is not enough for the application.

The second option involves using an opto-isolator. This element consist on a LED and a phototransistor. This allows to rise the voltage in the input of the ESC device without contact between both elements.

However, with the intention of simplifying the electronic assembly and also taking into account that it is easier to program Atmega328p than the microcontroller included in Arduino Due, the final decision was to use Arduino Uno with timer 1 in fast PWM mode.

Chapter 4

System Identification

4.1 Open loop code

In an open loop system, the plant output is directly based on the system input instead on the error between the output and the reference. Open loop operation with a potentiometer that allows to choose the duty cycle directly is a previous step to identification tests that allows to verify the correct overall functioning of the system.

```
1
2   #include <avr/interrupt.h>
3   int channelA, channelB, Pulso, Angulo, cont, val;
4   float k, U, E;
5
6   void setup() {
7     Serial.begin(9600);
8     // BITS AND REGISTERS ASSOCIATED TO ENCODER
9       pinMode(2, INPUT);
10      pinMode(3,INPUT);
```

```
11        pinMode(9, OUTPUT);
12        attachInterrupt(0, Encoder1, CHANGE);
13        attachInterrupt(1, Encoder2, CHANGE);
14      //BITS AND REGISTERS ASSOCIATED TO INTERRUPTS (TIMER2)
15        cli();                  // Deactivating global interruptions
16          TCCR2A = 0;
17          TCCR2B = 0;
18        TCCR2A =  (1<<WGM21);   // Comparator mode
19          OCR2A = 156;
20          TIMSK2 = (1 << OCIE2A);  // Activating interruption
21          TCCR2B = 0x07;                  // Prescaling at 1024
22        sei();                  //Global interruption activation
23      //BITS AND REGISTERS ASSOCIATED TO PWM (TIMER1)
24        TCNT1=0;              // clear counter
25        ICR1=31999;           // 500 Hz from 16 MHz clock
26      OCR1A=0;            // 10 % DC
27        TCCR1A=0b10000010; // non-inverting, fast PWM
28      TCCR1B=0b00011001; // fast PWM, full speed
29    }
30
31    void loop() {
32        val=analogRead(0);
33        OCR1A=map(val,0, 1023, 0, 65535);
34    }
35
36    void Encoder1() {
37        if (digitalRead(3)==digitalRead(2)){Pulso++;}
38        else{Pulso--;}
39      }
```

```
40
41    void Encoder2() {
42        if (digitalRead(3)==digitalRead(2)){Pulso--;}
43        else{Pulso++;}
44      }
45
46    ISR(TIMER2_COMPA_vect) {
47        cont ++;
48        if (cont == 2){
49            cont=0;
50            Serial.println(Pulso);
51            Serial.println(val);
52            Serial.println(OCR1A);
53        }
54     }
```

The first part of the project is devoted to the open loop configuration that allows to sample the input and output for a following model identification. The program meet these requirements:

- **Manual selection of the duty cycle with a potentiometer.** Pin 3 is defined as an analog input for the potentiometer. In void loop, which is repeated continuously, the value read with the command *analogRead* is assigned to variable *val*. The signal range is 0 to 5 volts and it is converted to an integer between 0 and 1023. This means that if duty cycle is selected by an analog input, there will be a loss of resolution when the value is scaled (10 bit input-16 bit output).

- **PWM signal output to ESC.** In Arduino Uno boards a 500Hz PWM signal

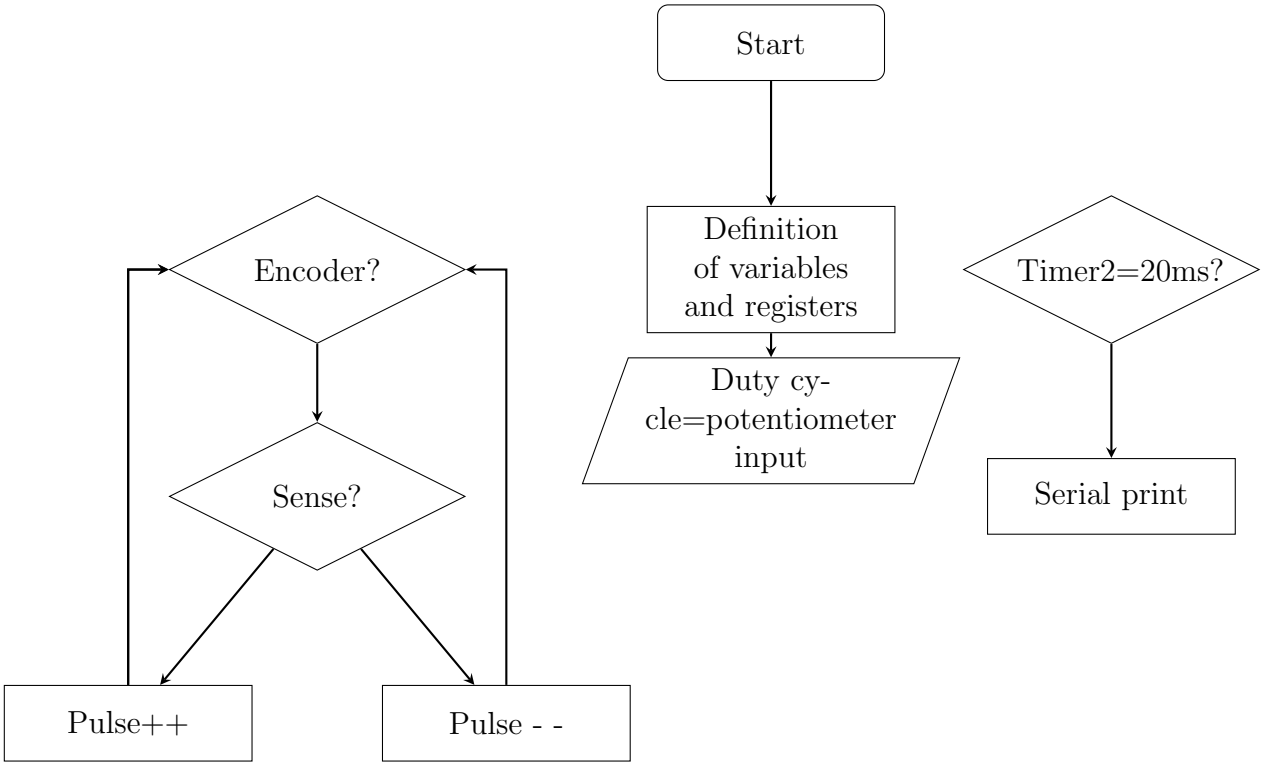


Figure 4.1: Open loop flowchart.

can be obtained from pins 9 and 10. In this example pin 9 is chosen, labeled as *motor* and defined in the setup void as an output. Register *OCR1A* regulates the duty cycle with a value between 0 and 65535 (16 bit resolution). As a consequence, the measurement from the potentiometer needs to be scaled with the command *map*.

- **Reading of the position by means of the encoder.** As said before, there are two signals coming from the encoder for which digital pins 2 and 3 are declared as inputs. Given that these signals present an square pulse for each slot, an interruption (*void Encoder*) function is enabled when either of the two pins detects a edge. The mechanism for the interrupt function is explained in section 2.4.1.

  It is necessary to take into account that the encoder is incremental, and therefore the 0^o point is set in the moment the code is launched. However, the starting point is usually the lower equilibrium point, so it is convenient to take is as a reference.

- **Sampling.** The objective of this open loop configuration is to obtain data of the input (PWM signal) and output (position). For a correct identification, these samples must be measured with a very precise sampling time. Besides, they must be sent to a PC through serial communication for later analysis. Choosing the communication speed must therefore be the first part of the sampling code:*Serial.begin(9600)*

  The use of delays can not ensure an exact and constant sampling time, for it will depend on the execution time of the code, variable with the result of the if

structures and the call of interruption voids. For this reason, an interruption function using timer 2 is the best option.

4.2 Identification

System identification is a method to build mathematical models of dynamic systems out of measured data. An important part of the process consists on designing experiments to obtain the required input-output data. When no model is available the method is called **black box** identification. If there is a model, for instance, the physical laws are known but there are parameters to be calculated, identification is performed through **grey box** modeling.

The first test for the data acquisition was carried out before the resolution problem was solved. As a consequence, duty cycle values are around 218 (where 100% corresponds to 255), a positive step to 222 and a negative step to 215 make the system oscillate around the horizontal equilibrium point. Sampling period was 20ms and the the code used was a variation of the open loop code only differing in the *void loop* function (Figure 4.2).

```
1  void loop() {
2
3    analogWrite(9,222);
4    delay(500)
5    analogWrite(9,215);
6    delay(1000)
7    analogWrite(9,218);
8    delay(8000)
9
10 }
```

Figure 4.2: Void loop for identification test

Data was obtained by means of the serial monitor and plotted using Matlab.

The fact that the mean value have been subtracted must be taken into account while reading data from figure 4.3.

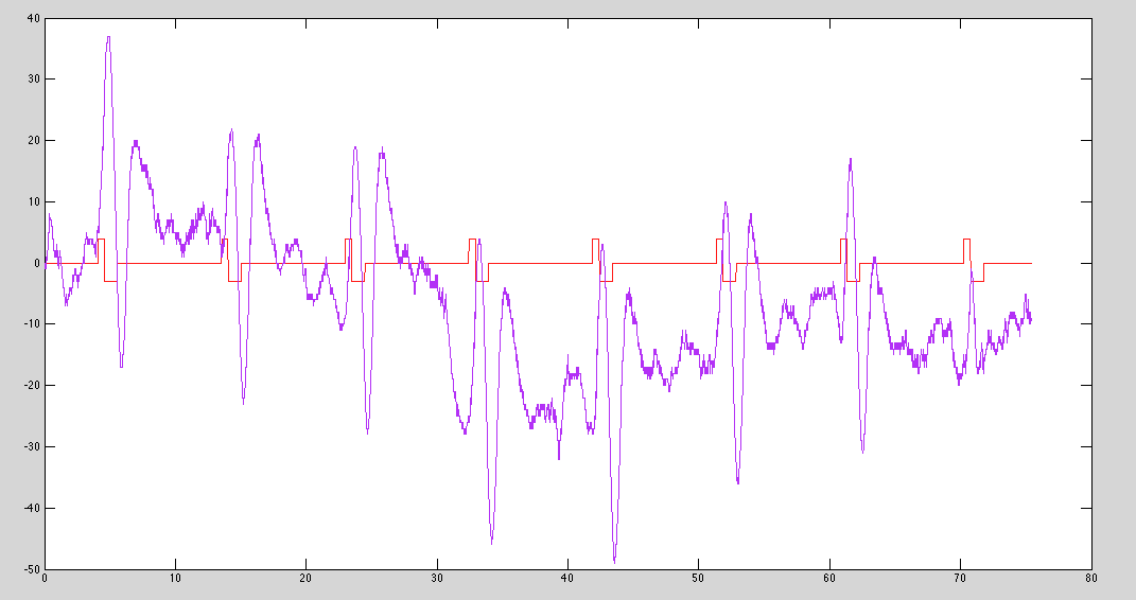


Figure 4.3: Test 1

The desired plant model will be a second order, which will hold the following expression:

$$C(z) = k\frac{\omega_n^2}{s^2 + 2\zeta\omega_n + \omega_n^2} \tag{4.1}$$

Once the first controllers were implemented, it has been found that the operation is very good in angles close to the lower equilibrium point (0 degrees) but while getting closer to the horizontal oscillation around the equilibrium point appears. According to the information about the control action and the error measured by the sensor, it is possible to infer that there is no reading or calculating mistake and neither a resolution one. This means that the controller should be improved.

The systems response depends on the operation point as a result of the non

linearity of the system. Figures 4.4 and 4.5 show how for a same amplitude step input, responses vary corresponding to the position.

Figure 4.4 shows how for a 1000 amplitud step the response corresponding to a higher angle (purple) has a greater peak in the falling edge, although the peak in the rising flank is almost the same. Figure 4.5 shows a greater effect of the position whilst introducing a 500 units amplitud step input. The red line corresponds to a small angle (around 12 degrees) and shows no oscillations. The higher the angle (green corresponds to 25 degrees and blue to 45), the greater the overshoot.

With this data, it would be possible to perform different identification tests for diverse position ranges, so that the plant model would be more accurate and thus, the controller would have a more effective role.

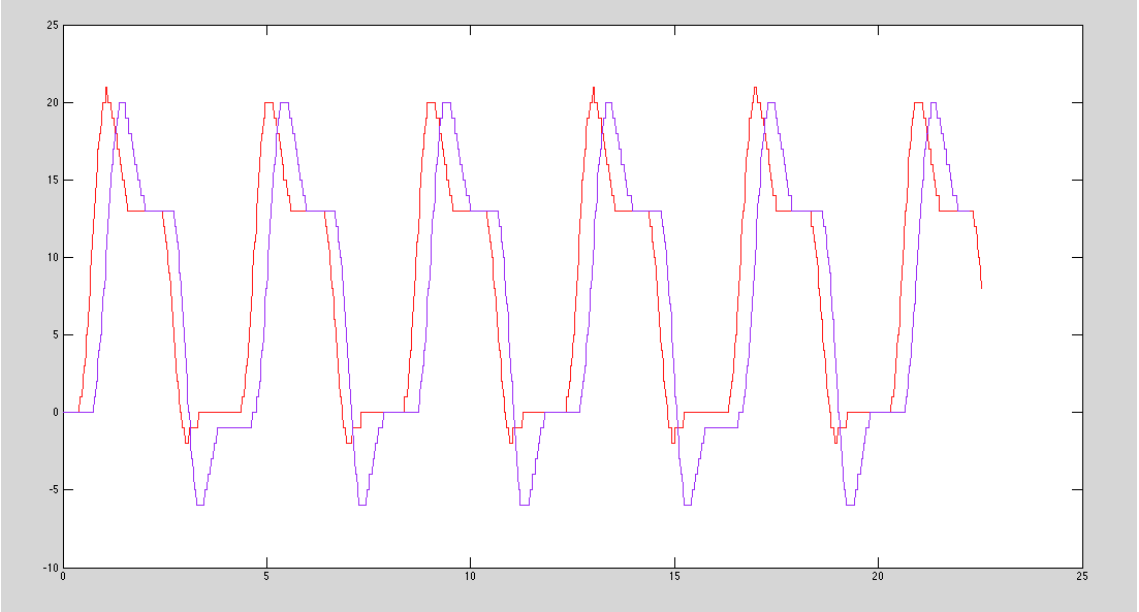


Figure 4.4: Test 2

$$G_{1b}(s) = 0.0129\frac{36}{s^2 + 7.2s + 36} * e^{-0.1s} \tag{4.2}$$

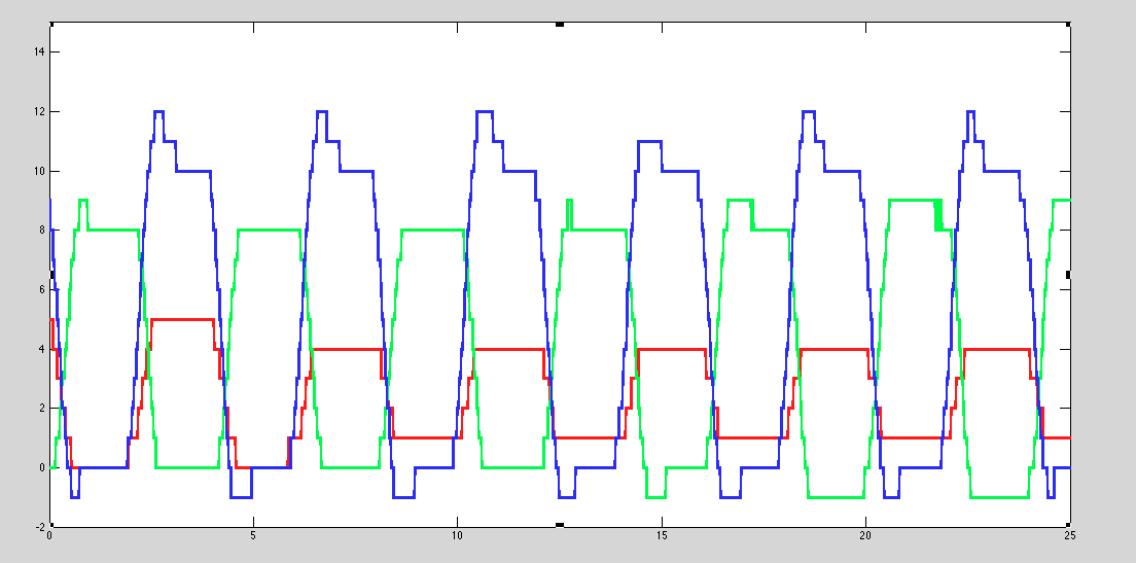


Figure 4.5: Test 3

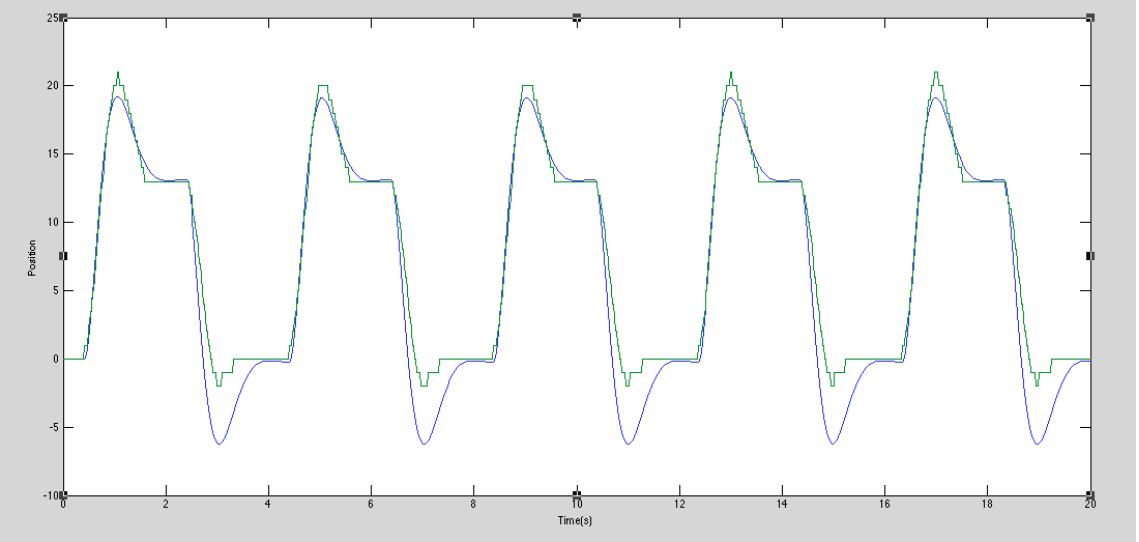


Figure 4.6: First test, fitting upwards movement

$$G_{1s}(s) = 0.0129\frac{25}{s^2 + 5.5s + 25} * e^{-0.1s} \tag{4.3}$$

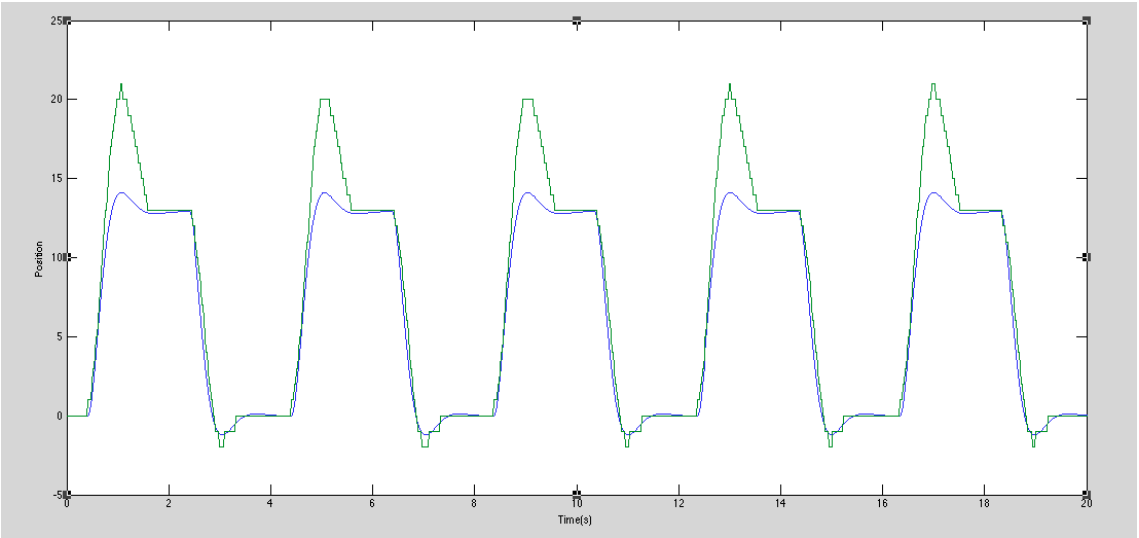


Figure 4.7: First test, fitting downwards movement

$$G_2(s) = 0.006\frac{25}{s^2 + 8s + 25} * e^{-0.1s} \tag{4.4}$$

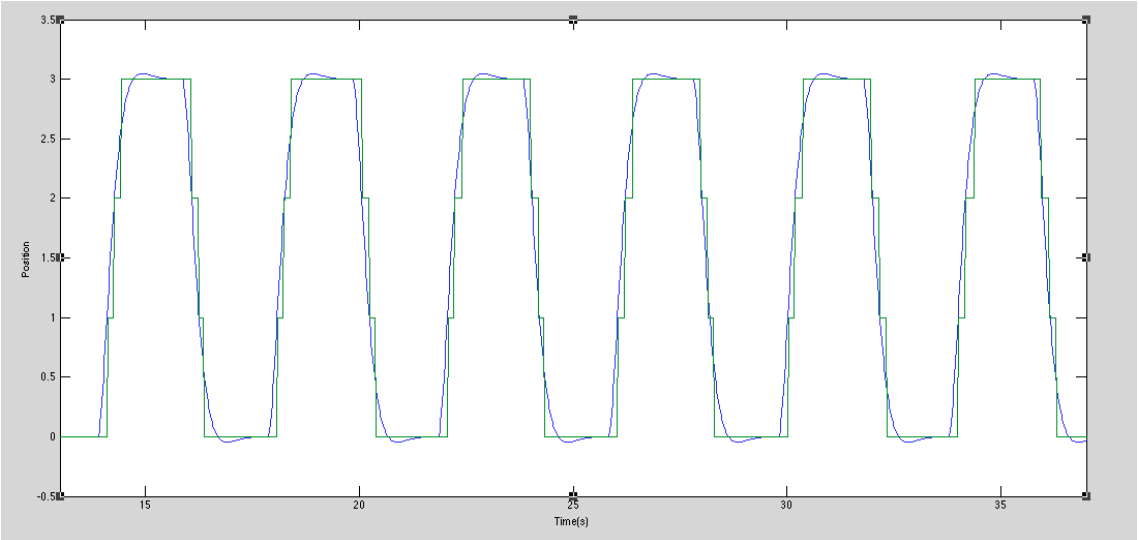


Figure 4.8: Second test

$$G_3(s) = 0.013 \frac{25}{s^2 + 5.5s + 25} \frac{\frac{s}{1.2} + 1}{\frac{s}{2} + 1} * e^{-0.1s} \tag{4.5}$$

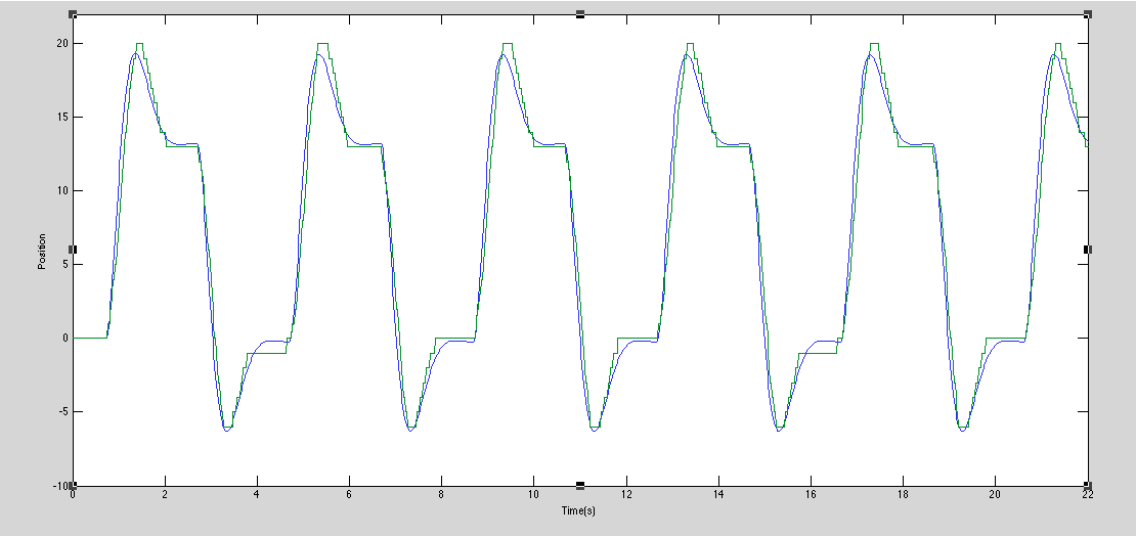


Figure 4.9: Third test

$$G_4(s) = 0.016 \frac{25}{s^2 + 7s + 25} \frac{\frac{s}{1.8} + 1}{\frac{s}{2} + 1} * e^{-0.1s} \tag{4.6}$$

$$G_5(s) = 0.02 \frac{25}{s^2 + 6.5s + 25} \frac{\frac{s}{1.5} + 1}{\frac{s}{2} + 1} * e^{-0.1s} \tag{4.7}$$

$$G_6(s) = \frac{0.05578s^2 - 0.01065s + 0.0003402}{s^4 + 0.4541s^3 + 2.622s^2 + 0.2942s + 0.06137} \tag{4.8}$$

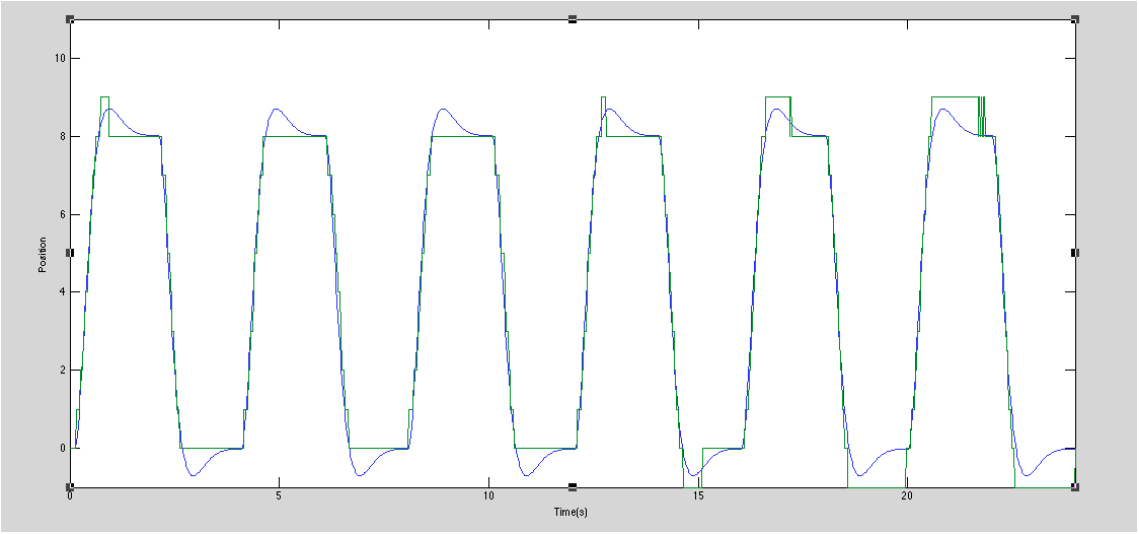


Figure 4.10: Forth test

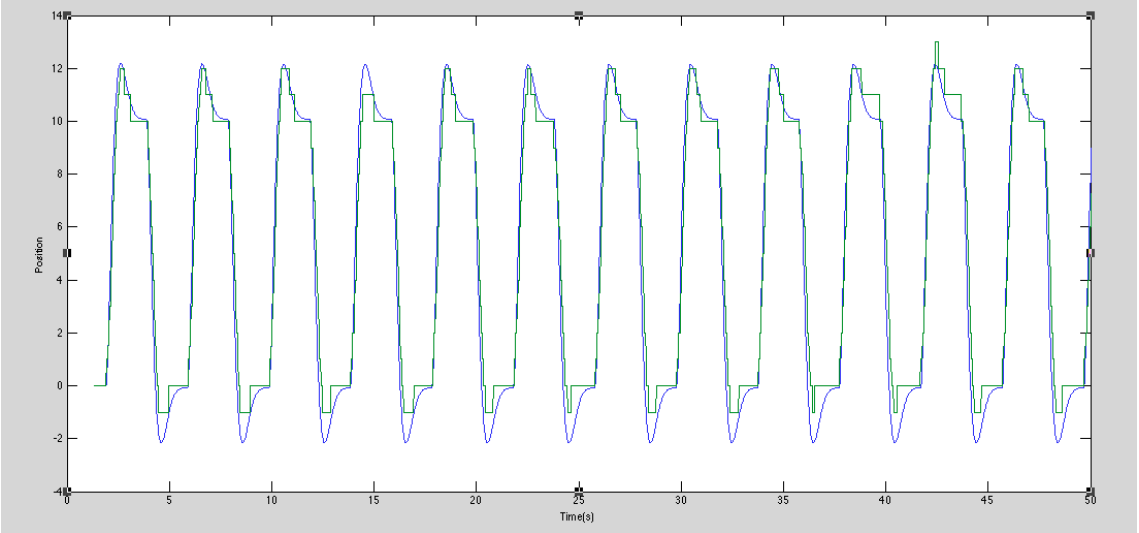


Figure 4.11: Fifth test

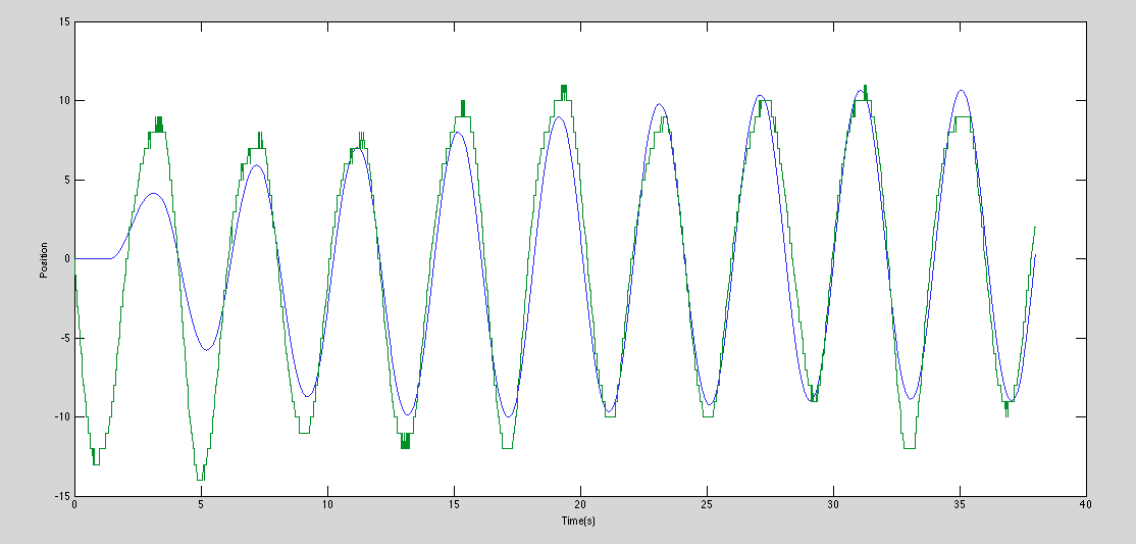


Figure 4.12: Sixth test

Chapter 5

Control

Once the controller has been calculated, it should be transformed into code lines in order to implement it in Arduino. Discrete transfer functions can be directly written as difference equations, which allow the calculation of the new control action in terms of the current error past actions and error (the minimum number of needed data depends on the degree of the controller).

Setting as an example the controller:

$$C(z) = \frac{12.02z - 11.32}{z - 1} \tag{5.1}$$

and taking into account that the control loop has the following diagram:

where **E(s)** is the error between the reference and the signal measured by the sensor and **U(z)** is the action of the controller, the subsequent equations hold:

$$C(z) = \frac{U(z)}{E(z)} = \frac{12.02z - 11.32}{z - 1} \tag{5.2}$$

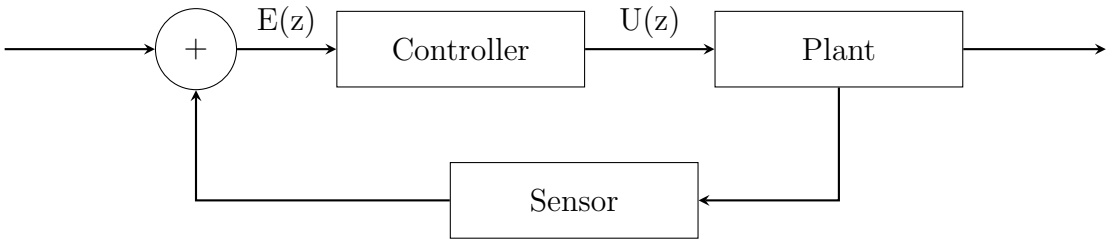


Figure 5.1: Closed loop.

$$E(z) * (12.02z - 11.32) = U(z) * (z - 1) \tag{5.3}$$

which could also be written as:

$$E(z) * (12.02 - 11.32z^{-1}) = U(z) * (1 - z^{-1}) \tag{5.4}$$

z^{-1} is a delay operation so the equation could be expressed as:

$$12.02 * E(n) - 11.32 * E(n-1) = U(n) - U(n-1) \tag{5.5}$$

$$U(n) = U(n-1) + 12.02 * E(n) - 11.32 * E(n-1) \tag{5.6}$$

This means that in order to write the corresponding code, it will be necessary to store three variables ($E0 = E(n-1)$, $E1 = E(n)$ and $U0 = U(n-1)$

Sampling period equals 20ms. During this time the sensor must read the new position, U0, E1 and E0 must be updated and the new value of U1 should be written in register **OCR1A** after its calculation.

Besides of implementing the controller, this code allows changing the angle ref-

erence without having to change the code and reprogram the board. To enable that, it is necessary to use the serial monitor.

While there is no input to the serial port, Arduino reads a constant value of -1. Therefore, whenever it reads a value over 0, it will be assigned to the variable that represents the reference in the control loop.

```
#include <avr/interrupt.h>
int channelA, channelB, Pulso, Angulo, cont, val;
float E2, E1, E0, U2, U1, U0, U;
 int T=0.02;
int ref, posfin, incomingByte, newval;
void setup() {
    Serial.begin(9600);
    ref=0;
  //REFERENCE
  Serial.println("Valor de la referencia");
  while(Serial.available()==0){}
  posfin=Serial.parseInt();
  //ENCODER BIT AND REGISTER DEFINITION
    pinMode(2, INPUT);
    pinMode(3,INPUT);
    pinMode(9, OUTPUT);
     attachInterrupt(0, Encoder1, CHANGE);
     attachInterrupt(1, Encoder2, CHANGE);
  //Timer 2 BIT AND REGISTER DEFINITION
  cli();
  TCCR2A = 0;
  TCCR2B = 0;
  TCCR2A =(1<<WGM21);
  OCR2A = 156;
```

```
25    TIMSK2 = (1 << OCIE2A);
26    TCCR2B = 0x07;
27    sei();
28    //PWM BIT AND REGISTER DEFINITION
29    TCNT1=0;
30    ICR1=32001;          // 500 Hz from 16 MHz clock
31    OCR1A=0;            // 10 % DC
32    TCCR1A=0b10000010; // non-inverting, fast PWM
33    TCCR1B=0b00011001; // fast PWM, full speed
34  }
35  void loop() {
36      while(Serial.available()==0){
37      Serial.print(Pulso);
38      Serial.print("  ");
39      Serial.print(E1);
40      Serial.print("  ");
41      Serial.println(U1);
42    }
43    if(newval==0){
44      Serial.println(Pulso);
45    }
46    if(newval!=0){
47      posfin=newval;
48    }
49  }
50  void Encoder1() {
51    if (digitalRead(3)==digitalRead(2)){
52      Pulso++;
53    }
```

```
54      else{
55      Pulso--;
56    }
57 }
58 void Encoder2() {
59      if (digitalRead(3)==digitalRead(2)){
60      Pulso--;
61    }
62      else{
63      Pulso++;
64    }
65 }
66 ISR(TIMER2_COMPA_vect) {
67      cont ++;
68      if (cont == 2){
69      cont=0;
70      E0=E1;
71      U0=U1;
72      E1=posfin-Pulso;
73      U1=U0+12.02*0.1*E1-11.32*0.1*E0;
74      OCR1A=0;
75    }
76 }
```

Close loop code could be explained with the flowchart diagram 5.2.

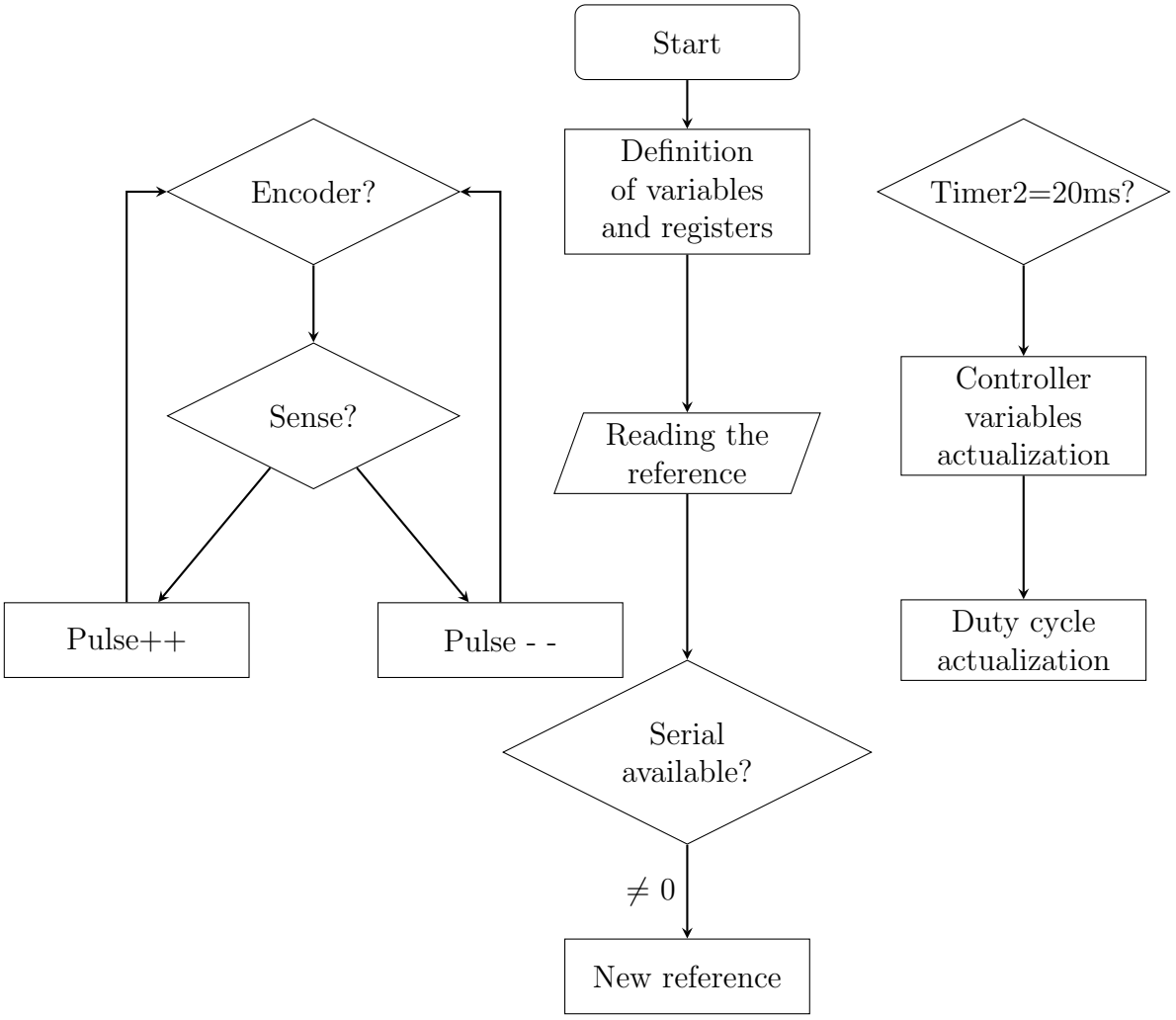


Figure 5.2: Closed loop flowchart.

As happens in the open loop configuration, it is necessary to use interrupts to ensure that operations follow the correct order. The value of variable **Pulso** is actualized every time Arduino detects an external interrupt. Then, when timer 2 reaches **OCR2A** value, another interrupt function is invoked and all variables involved in the calculation of the action are renewed. After that, a new value is written into **OCR1A** and the duty cycle changes. This process is repeated every

sampling period.

5.1 Controlers

Although, for angles below 90^o the plant it is not necessary to stabilize the the system, feedback allows to dump or eliminate perturbation, have a better tracking of the reference and eliminate the steady state error by using integral action.

5.1.1 Controller 1

$$C(z) = \frac{12.02z - 11.32}{z - 1} \tag{5.7}$$

In order to reduce oscillation without making the system slower in the lower point, the previous controller is multiplied by a changing parameter function of the position, that obeys the following formula:

$$k = -0.327k_c + 0.968 \tag{5.8}$$

As a result, the coefficient that multiplies the controller is:

Rango (o)	k
0-30	0.968
30-60	0.641
60-90	0.314

Table 5.1: Values for constant k

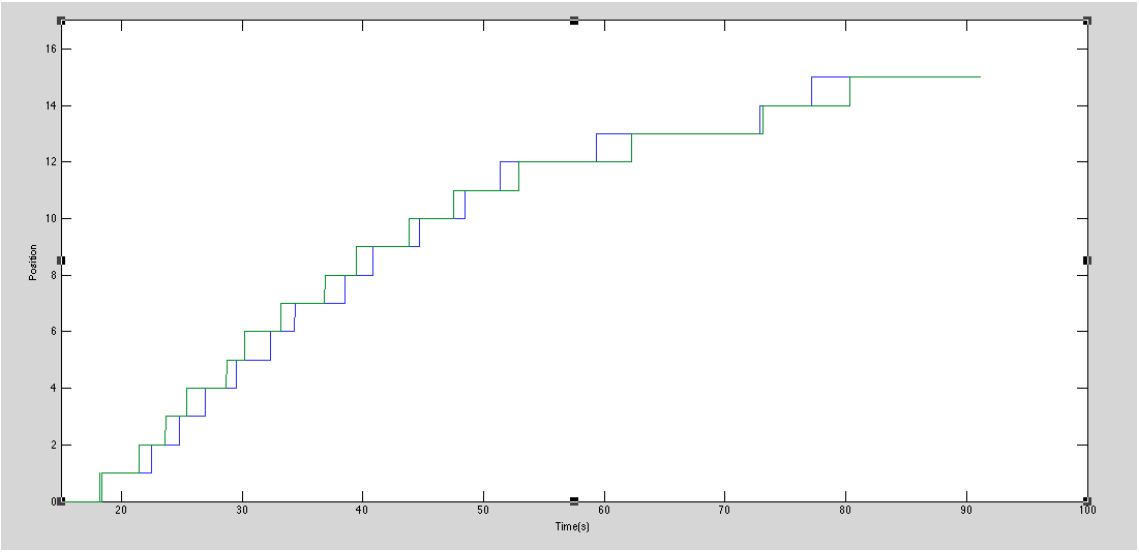


Figure 5.3: Controllers 5.7 and 5.8, 0^o to 15^o

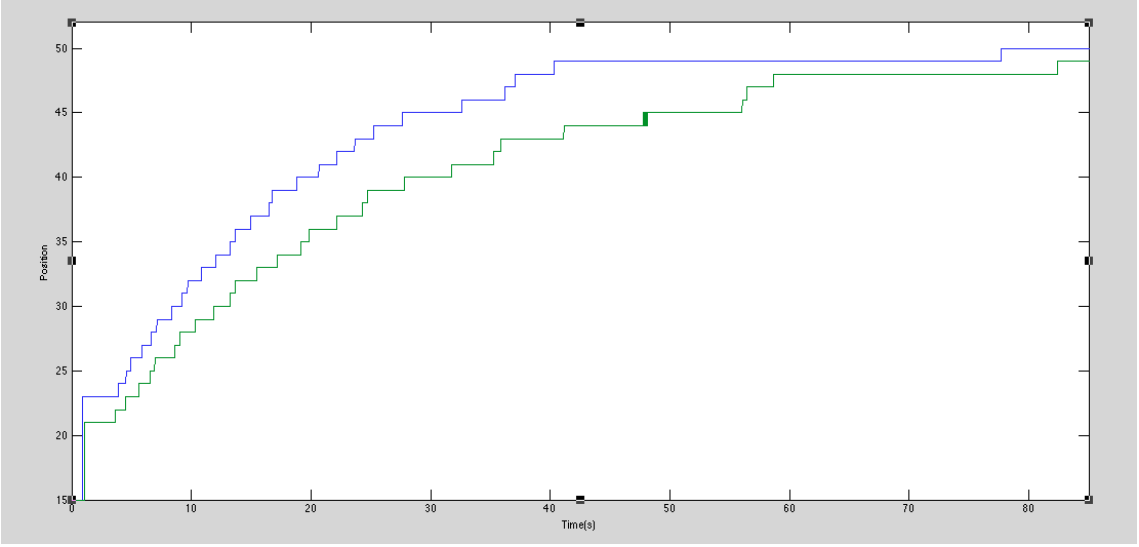


Figure 5.4: Controllers 5.7 and 5.8, 15^o to 50^o

Figures 5.3 and 5.4 depict the output of the system when controller 5.7 is applied

(blue line) and when the position of the zero and the gain are changed through expression 5.8. For small angles, the formula for the controller is almost the same in both cases, and so is the system response. In the second angle range (figure:5.4) the value of k has changed, and so the second controller is slower.

5.1.2 Controller 2

This PI controller is designed for the more unstable angle range. As a consequence the integral constant **Ti** has a higher value and the gain is smaller.

$$C(s) = \frac{5s + 2.5}{s} \tag{5.9}$$

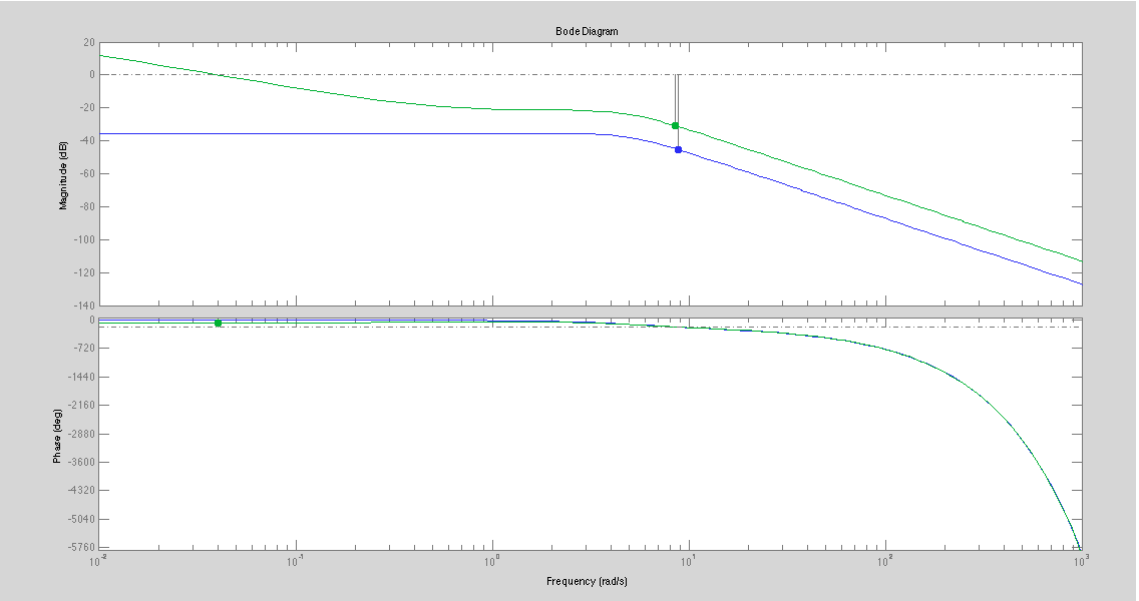


Figure 5.5:

Figure 5.5 shows stability margins difference between plant 4.7 identified for

middle angle values (green line) and the same plant with the controller(blue line).

Controlers are obtained through Tustin method for different sampling times:

T=0.02s

$$C(z) = \frac{5.025z - 4.975}{z - 1} \tag{5.10}$$

T=0.05s

$$C(z) = \frac{5.063z - 4.938}{z - 1} \tag{5.11}$$

T=0.2s

$$C(z) = \frac{5.25z - 4.75}{z - 1} \tag{5.12}$$

In the first test the reference is a step that changes from 0 to 15. Figure 5.6 shows the system response. Apart from the small overshoot for the controller with period T=0.2s, the response is very similar for the three of them.

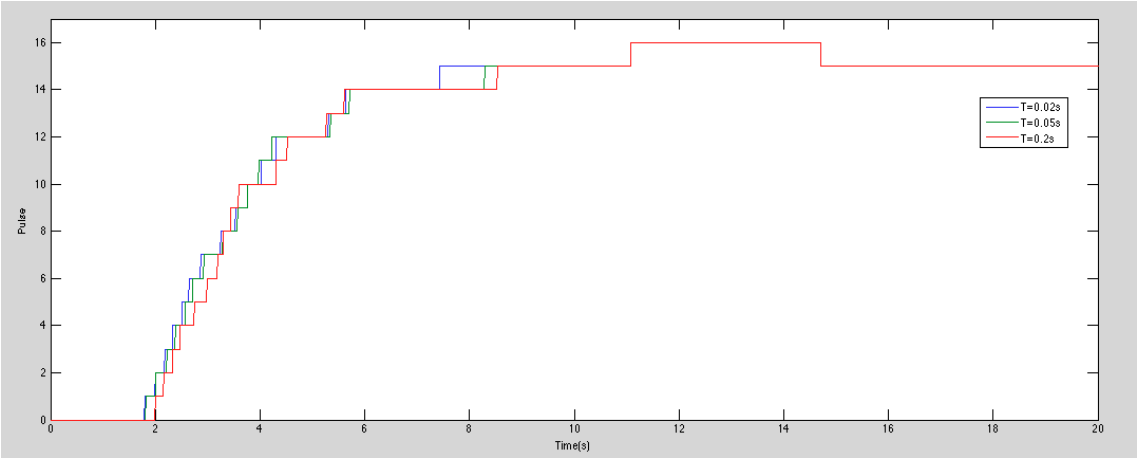


Figure 5.6: 0^o to 15^o controller 2

The following test consisted on a step with the amplitude changing from 15 to 50, the corresponding output is observed in 5.7.

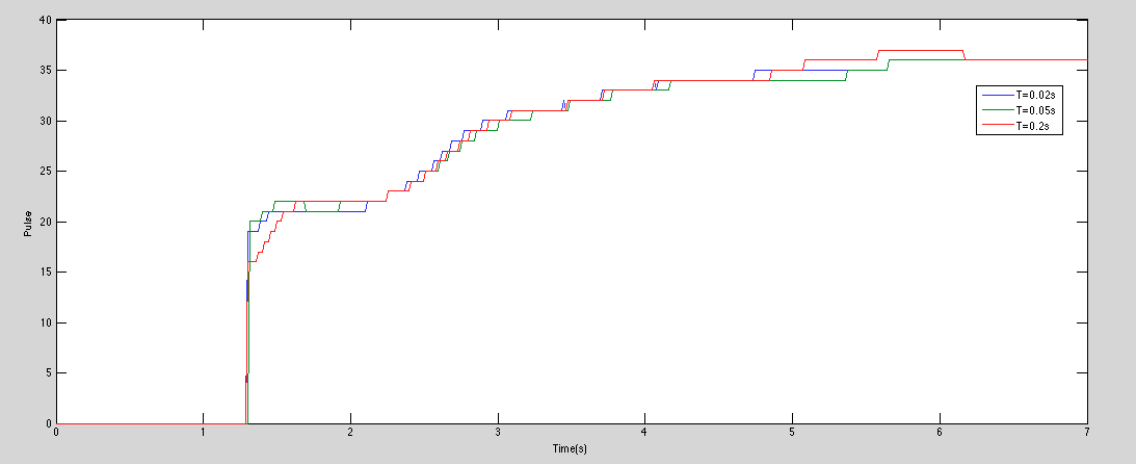


Figure 5.7: 15^o to 50^o controller 2

This controller only offered measurable results for 0-15 and 15-50 intervals. For higher values of the angle, the system is too slow to compensate oscillations above 90^o.

5.1.3 Controller 3

The position of the zero in this PI controller is closer to the origin making it faster.

$$C(s) = \frac{6.65s + 35}{s} \tag{5.13}$$

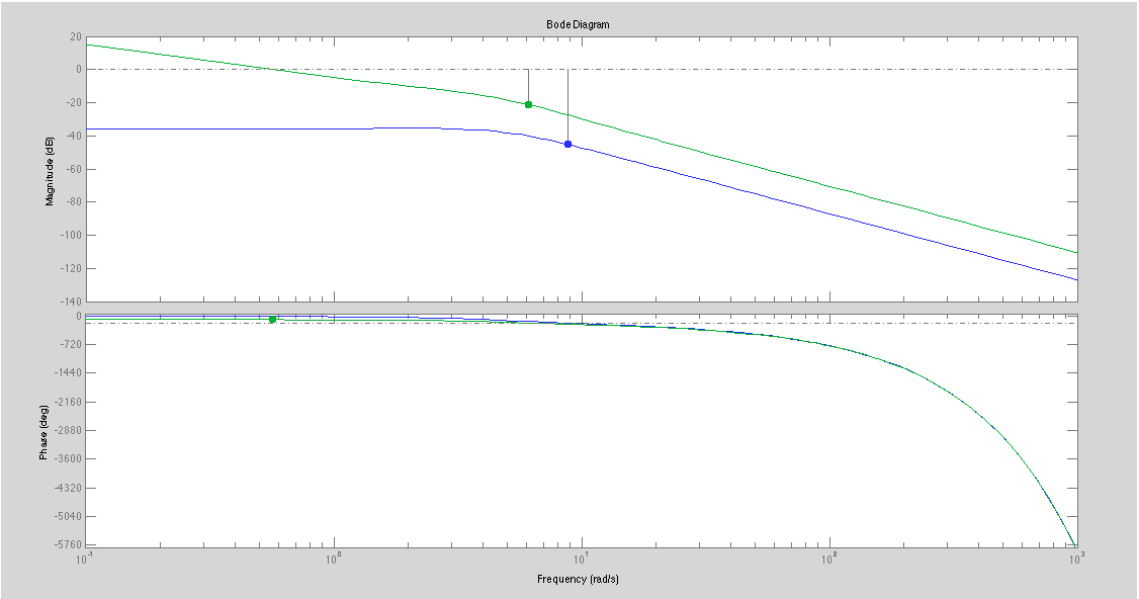


Figure 5.8:

Figure 5.8 shows stability margins difference between plant 4.7 identified for middle angle values (green line) and the same plant with the controller(blue line). It is interesting to notice the difference between stability margins using controller 5.13 depicted in figure 5.8 and the ones obtained by applying controller 5.9 shown in picture 5.5.

Discrete controllers obtained with Matlab using Tustin method for different periods are:

T=0.02s

$$C(z) = \frac{7z - 6.3}{z - 1} \tag{5.14}$$

T=0.05s

$$C(z) = \frac{7.875z - 6.125}{z - 1} \tag{5.15}$$

T=0.20s

$$C(z) = \frac{10.5z - 3.5}{z - 1} \tag{5.16}$$

Following figures show the measured data of the output for different angle ranges and periods.

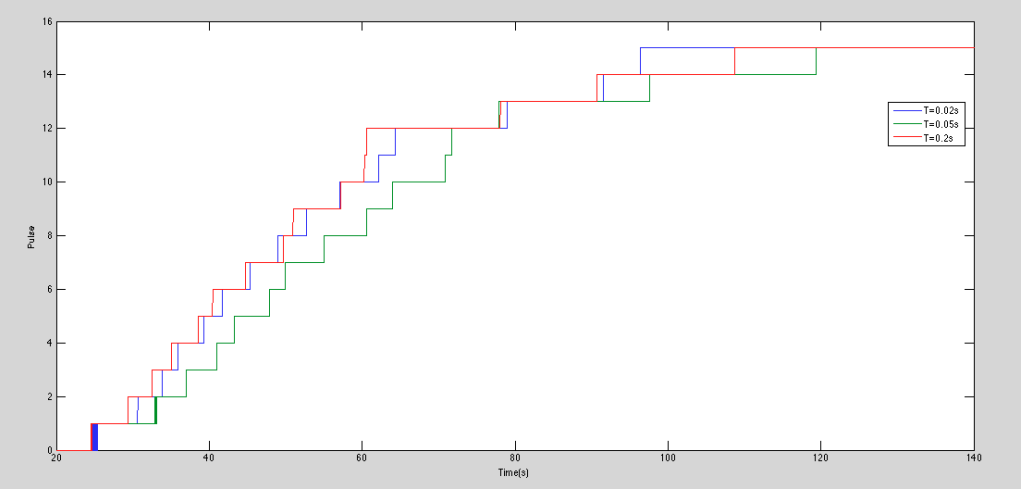


Figure 5.9: 0^o to 15^o controller 3

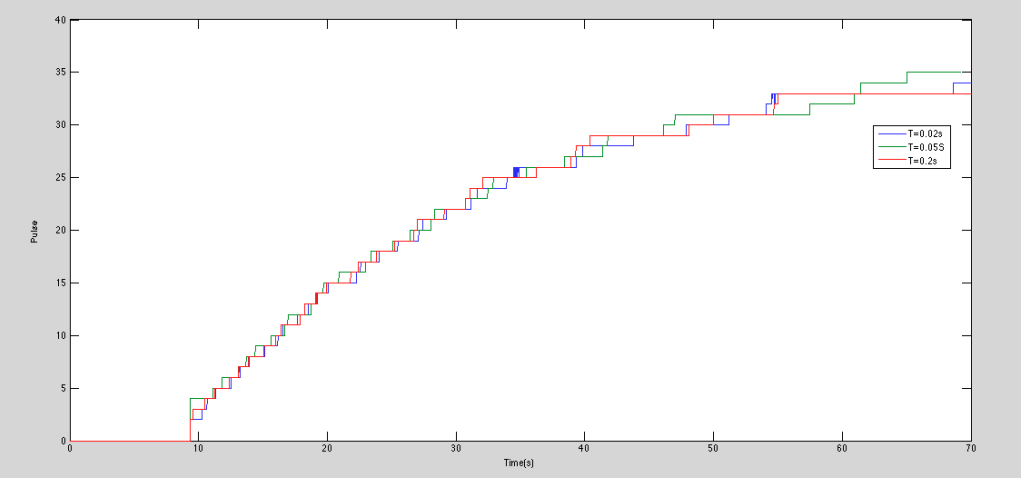


Figure 5.10: 15^o to 50^o controller 3

In the most stable zone, response is similar for the three different periods. The main difference with the controller applied in the last section is that for higher values of the angle, the system is still stable, although with significative oscillation.

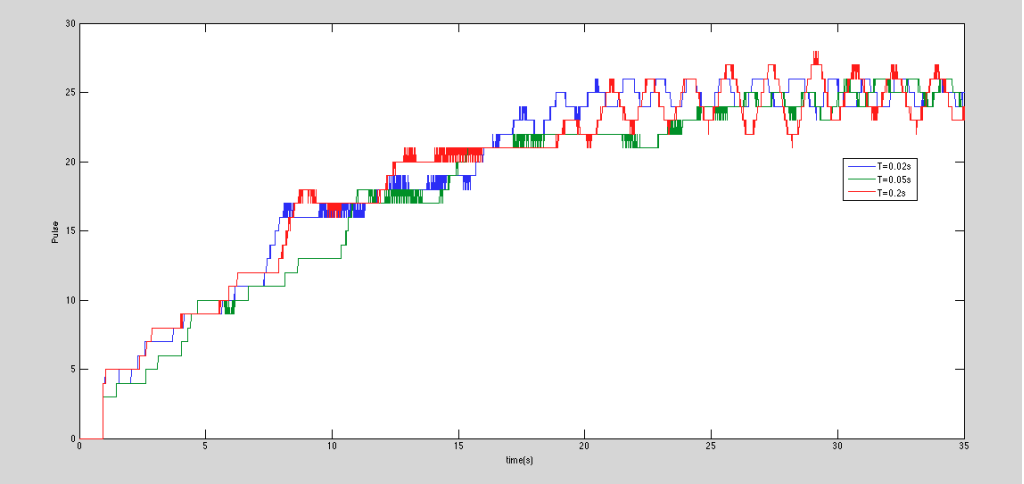


Figure 5.11: 50^o to 75^o controller 3

5.1.4 Linear interpolation between two controllers

With the aim of obtaining a controller that provides an acceptable result regardless of the position, the last controller presents a linear interpolation for the parameters that characterize the position of the zero and the gain of the controller.

The structure of the analog controller is:

$$C(z) = \frac{k(\frac{s}{c} + 1)}{s} \tag{5.17}$$

where the position of the zero and the gain change through the following expressions:

$$k = 35 - 32.5 * (1 - \frac{Pulso}{100}) \tag{5.18}$$

$$c = 5 - 4.5 * (1 - \frac{Pulso}{100}) \tag{5.19}$$

Therefore, at angles close to the lower point, the behavior will be similar to controller 5.13 and with higher values of the angle, the gain and the zero will get closer to the ones in 5.9. A digital controller is obtained by Tustin method, that uses the following approximation:

$$z = e^{sT} = \frac{1 + s\frac{T}{2}}{1 - s\frac{T}{2}} \tag{5.20}$$

$$z = \frac{z-1}{z+1}\frac{2}{T} \tag{5.21}$$

The result is shown in the subsequent equation:

$$C(s) = \frac{k(0.01 + \frac{1}{c})z - k(\frac{1}{c} - 0.01)}{z - 1} \tag{5.22}$$

Figures , and show the difference between this last controller and the two previous ones (with a sampling time of T=0.02s) for input steps of different amplitude and starting from different points.

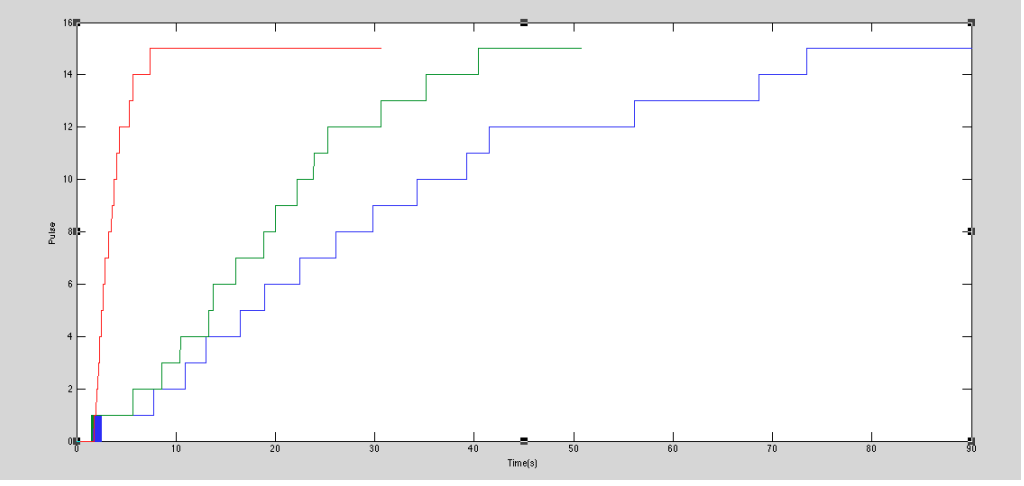


Figure 5.12: 0^o to 15^o, linear interpolation

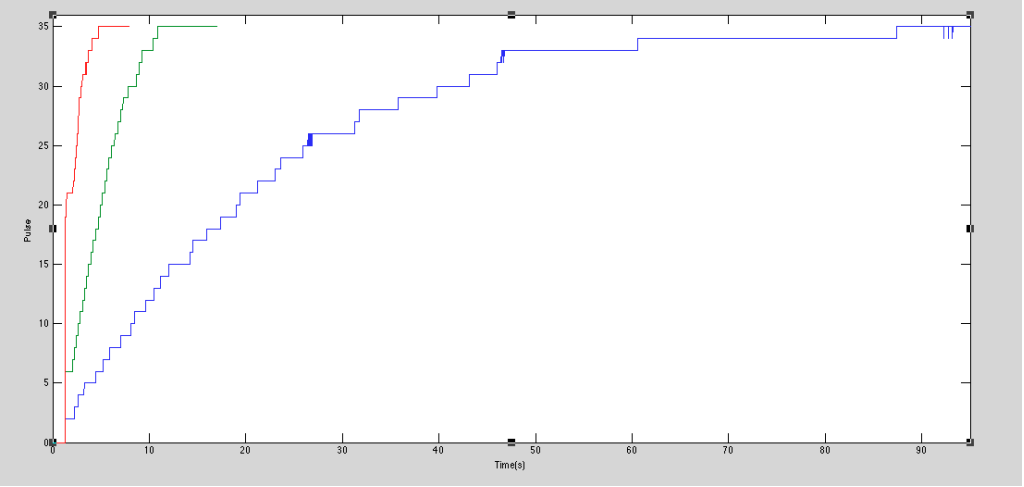


Figure 5.13: 15^o to 50^o, linear interpolation

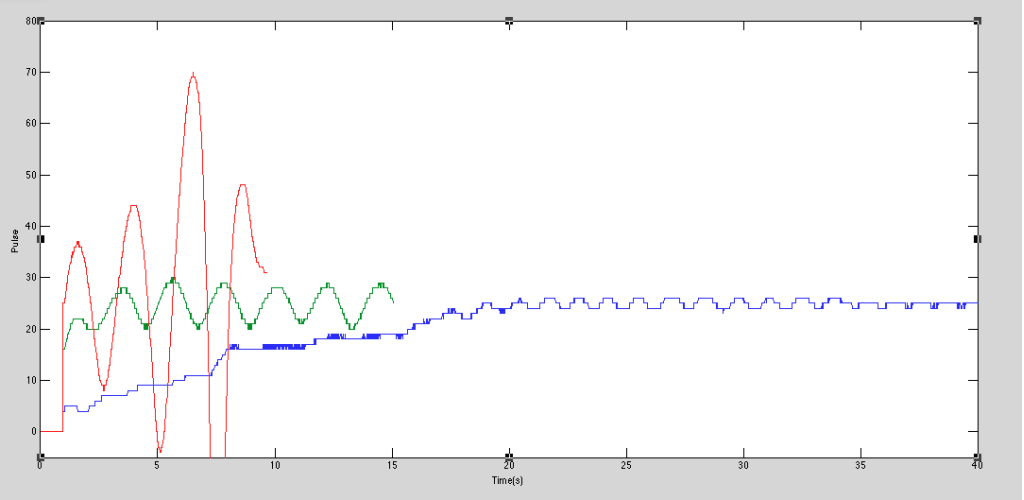


Figure 5.14: 50^o to 75^o, linear interpolation

5.2 Simulation vs acquired data.

The following figures show a simulation of the behavior of the designed systems and the measured data. The most important difference between simulation and real response is that the second is faster for every controller and chosen identified plant.

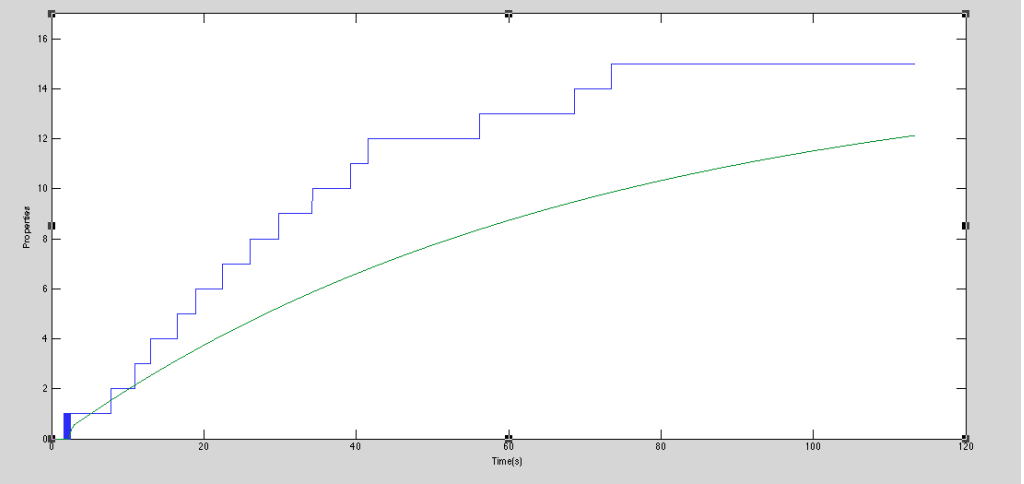


Figure 5.15: Controler 5.9, plant 4.5

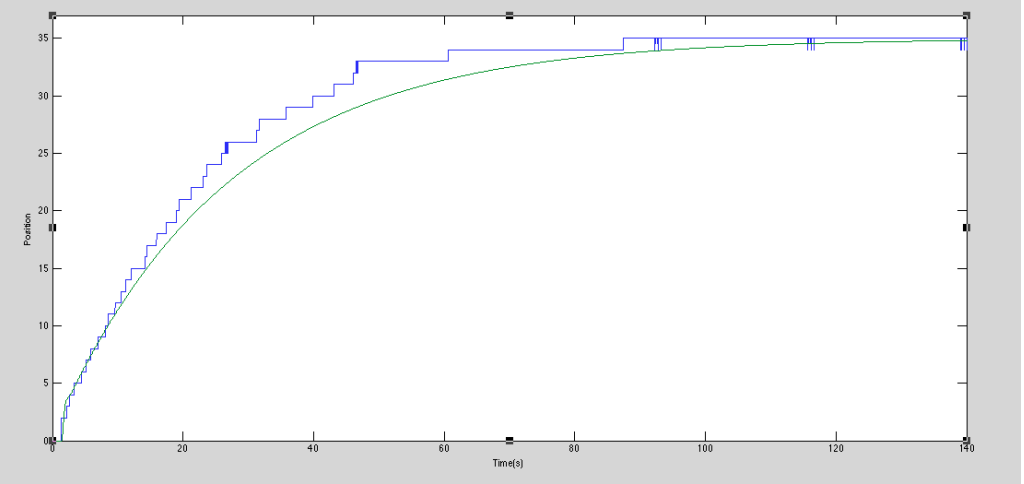


Figure 5.16: Controler 5.9, plant 4.7

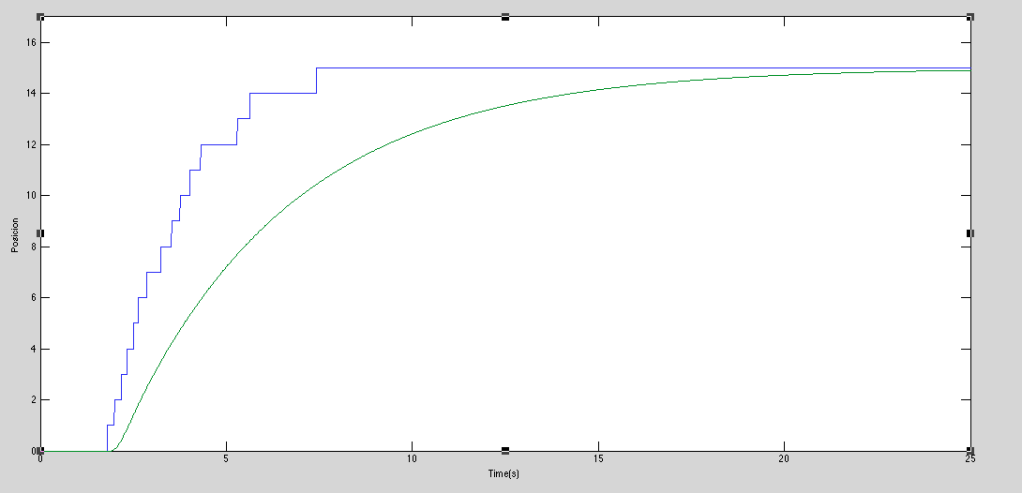


Figure 5.17: Controler 5.13, plant 4.5

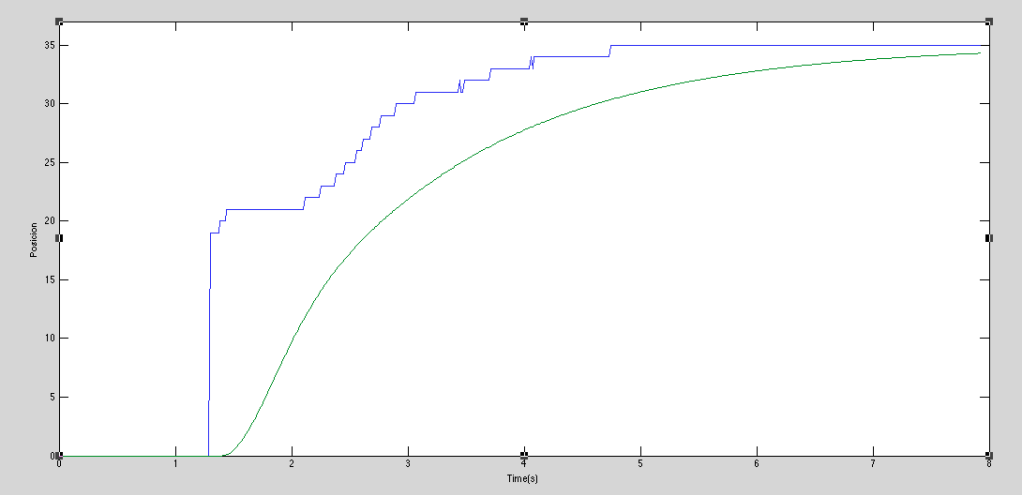


Figure 5.18: Controler 5.13, Plant 4.7

In the previous pictures it is possible to appreciate the difference in the speed of the controllers.

Chapter 6

Conclusions

The main part of the project focuses on selecting the tools to perform the control. The combination of the brushless motor and the ESC device has proved to be very convenient for the application. Although the initial sensor was not suitable in this case, the problem was not it the measuring technology but in the speed and overall features of the microcontroller.

Even though after reducing the resolution and optimizing the code the result was satisfactory, one must take into account that for more complicated systems Arduino Uno is not powerful enough and would have problems tracking the times.

Identification and controller design and implementation have been successful in the stable angle range. As shown in the previous sections, it is possible to introduce reference values through the serial monitor without having to program Arduino again and for angles below 90^o control works perfectly.

Further work lines could be performing a more accurate identification of the system in the unstable zone or the development of a Labview program that allows to monitor the state of the system in a computer and store data.

Bibliography

[1] Azcona, Ekaitz, *Diseño, Construcción y Control de un Helicóptero de un Grado de Libertard*, Universidad Pública de Navarra, 2015.

[2] Catalogue of *EMAX*, Home, BL Series Motors, BL2210, available in: `http://www.emaxmodel.com/brushless-motor/bl-series-motors/bl2210.html`.

[3] Catalogue of *EMAX*, Home, ESC, Simon K Series ESC, EMAX Simon Series 30A for multi-copter, available in: `http://www.emaxmodel.com/esc/simonk-series/emax-simon-series-30a-for-muti-copter.html`.

[4] Catalogue of Hengstler, Products, Rotatory encoders, Incremental rotatory encoders, Icuro RI32,`http://www.hengstler.de/en/s_c10030217/Rotary_encoders/Incremental_rotary_encoders/ICURO_RI32/`, .

[5] User:Cybergibbons,(21 of February of 2014) *Would an infinite loop inside loop perform faster?* `http://arduino.stackexchange.com/questions/337/would-an-infinite-loop-inside-loop-perform-faster`

[6] National Instruments, *Quadrature Encoder Fundamentals*, Página Principal, Productos y Servicios, Notas Técnicas, Quadrature Encoder Fundamentals `http://www.ni.com/white-paper/4763/en/`.

[7] ATMEL, *Atmel 8-bit microcontroller with 4/8/16/32kbytes in.system programmable flash datasheet*http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf, 2015.

[8] ARDUINO, https://www.arduino.cc.